

2

NAVAL POSTGRADUATE SCHOOL Monterey, California

DTIC FILE COPY

AD-A225 434



THESIS

DTIC
SELECTE
AUG 20 1990
Co E D

EPLD MODELING WITH VHDL

by

Shih-Ming Shu

December 1989

Thesis Advisor:

Chin-Hwa Lee

Approved for public release; distribution is unlimited

Unclassified

security classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified			1b Restrictive Markings		
2a Security Classification Authority			3 Distribution Availability of Report		
2b Declassification Downgrading Schedule			Approved for public release; distribution is unlimited.		
4 Performing Organization Report Number(s)			5 Monitoring Organization Report Number(s)		
6a Name of Performing Organization Naval Postgraduate School		6b Office Symbol (if applicable) 62	7a Name of Monitoring Organization Naval Postgraduate School		
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000			7b Address (city, state, and ZIP code) Monterey, CA 93943-5000		
8a Name of Funding Sponsoring Organization		8b Office Symbol (if applicable)	9 Procurement Instrument Identification Number		
8c Address (city, state, and ZIP code)			10 Source of Funding Numbers		
Program Element No		Project No	Task No	Work Unit Accession No	
11 Title (include security classification) EPLD MODELING WITH VHDL					
12 Personal Author(s) Shih-Ming Shu					
13a Type of Report Master's Thesis		13b Time Covered From To		14 Date of Report (year, month, day) December 1989	15 Page Count 135
16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.					
17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)		
Field	Group	Subgroup	VHDL,HDL,HARD WARE DESCRIPTION LANGUAGE,EPLD,PLA.		
19 Abstract (continue on reverse if necessary and identify by block number)					
<p>Incompatibility between separately-designed subsystems has long been a problem in the logic design industry. This problem greatly affects the productivity of logic design procedures. It also makes system maintenance and second source procurement very difficult. The military and IEEE 1076 standard hardware description language VHDL is a promising solution to this problem. In this thesis, the VHDL language was used to model an industry-wide popular device -- erasable programmable logic device (EPLD). The EPLD modeling problems are discussed via the modeling of two EPLD chips, EP310 and EP1800. The solutions to these problems are described and tested. The goal of this thesis is to provide examples of VHDL coding techniques related to the EPLD modeling. These coding techniques with the associated EPLD library can be used to support future system level logic design.</p>					
20 Distribution Availability of Abstract			21 Abstract Security Classification		
<input checked="" type="checkbox"/> unclassified unlimited <input type="checkbox"/> same as report <input type="checkbox"/> DTIC users			Unclassified		
22a Name of Responsible Individual Chin-Hwa Lee			22b Telephone (include Area code) (408) 646-2100		22c Office Symbol 621 e

DD FORM 1473,84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

EPLD Modeling with VHDL

by

Shih-Ming Shu

Lieutenant, Taiwan Republic of China Navy

B.S., Chinese Naval Academy 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

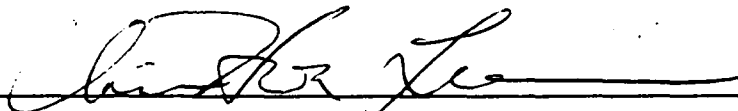
December 1989

Author:

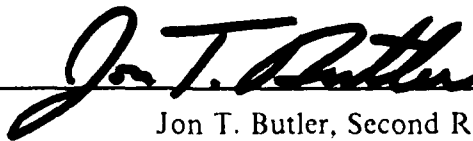


Shih-Ming Shu

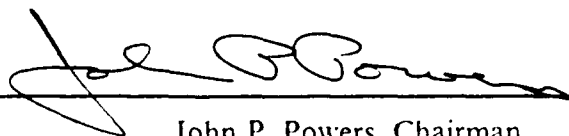
Approved by:



Chin-Hwa Lee, Thesis Advisor



Jon T. Butler, Second Reader



John P. Powers, Chairman,
Department of Electrical and Computer Engineering

ABSTRACT

Incompatibility between separately-designed subsystems has long been a problem in the logic design industry. This problem greatly affects the productivity of logic design procedures. It also makes system maintenance and second source procurement very difficult. The military and IEEE 1076 standard hardware description language VHDL is a promising solution to this problem. In this thesis, the VHDL language was used to model an industry-wide popular device -- erasable programmable logic device (EPLD). The EPLD modeling problems are discussed via the modeling of two EPLD chips, EP310 and EP1800. The solutions to these problems are described and tested. The goal of this thesis is to provide examples of VHDL coding techniques related to the EPLD modeling. These coding techniques with the associated EPLD library can be used to support future system level logic design.

Accession For	
NTIS	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
Fr	
Distribution/	
Availability Codes	
and/or	
Dist	Special
A-1	



THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

I. INTRODUCTION	1
A. VHSIC HARDWARE DESCRIPTION LANGUAGE (VHDL)	1
B. ERASABLE PROGRAMMABLE LOGIC DEVICE (EPLD)	2
C. EPLD MODELING	3
II. VHDL FEATURE AND ENVIRONMENT	5
A. BASIC FEATURES OF VHDL	5
1. DESIGN ENTITIES	5
2. CONCURRENT STATEMENTS	11
a. BLOCK STATEMENT	11
b. PROCESS STATEMENT	14
3. DATA TYPES	14
4. CLASS OF OBJECTS	19
a. Constants	19
b. Variables	19
c. Signals	20
5. ATTRIBUTES	22
6. SUBPROGRAM AND PACKAGES	23
7. OPERATORS AND CONTROL STATEMENTS	26
B. VHDL SUPPORTING ENVIRONMENT	28
1. SIMULATION PROCEDURE	30
2. REPORT GENERATOR	32
3. VHDL LIBRARY SYSTEM (VLS)	33

III. MODELING THE EP310	35
A. INTRODUCTION OF THE EP310	35
B. DEFINE THE PROBLEM	39
C. DECOMPOSITION OF THE EP310	39
D. ESTABLISH DATA FLOW	40
E. SIGNAL ASSIGNMENT	43
F. JEDEC FILE INFORMATION TRANSFER	44
G. MULTIPLE LEVEL LOGIC	46
H. BUSSED SIGNAL	47
I. PRODUCT TERM INTERNAL CONNECTION	49
J. EPLD TIMING SIMULATION	51
I. REGISTER TIMING	52
K. COUNTER	57
IV. MODELING THE EP1800	58
A. INTRODUCTION OF THE EP1800	58
B. DECOMPOSITION OF THE EP1800	61
C. ESTABLISH DATA FLOW OF THE MODEL	62
D. GENERAL MACROCELL AND ENHANCED MACROCELL	66
E. THE REUSABLE QUADRANT MODEL	66
F. THE EP1800 BUS STRUCTURE	68
G. UP DOWN COUNTER	69
V. CONCLUSIONS	71
A. GENERAL	71
B. PROGRAM SPEED	72

C. RECOMMENDATIONS FOR FUTURE STUDY	73
APPENDIX A. VHDL SOURCE CODE FOR EP310 AND EP1800 MODELS ..	74
A. VHDL SOURCE CODE FOR EPLD_PACKAGE	74
B. VHDL SOURCE CODE FOR EP310 MODEL	77
C. VHDL MODEL FOR EP1800.	85
APPENDIX B. VHDL CODE FOR TEST_BENCH	97
A. VHDL SOURCE CODE FOR TOP ENTITY DECLARATION.	97
B. TEST_BENCH ARCHITECTURE BODY FOR EP310	98
C. TEST_BENCH ARCHITECTURE BODY FOR EP1800	103
APPENDIX C. EXAMPLES OF SIGNAL SELECT FILE AND SIGNAL MAP	
FILE	108
A. SIGNAL SELECT FILE	108
B. SIGNAL MAP OF THE TEST_EP3 MODEL	109
APPENDIX D. MACRO VAX VMS SYSTEM COMMAND	119
A. MACRO VAX VMS SYSTEM COMMAND FOR EP310 MODEL	119
B. MACRO VAX VMS SYSTEM COMMAND FOR EP1800 MODEL ...	120
LIST OF REFERENCES	121
INITIAL DISTRIBUTION LIST	123

LIST OF TABLES

Table 1. ATTRIBUTE VALUES	23
Table 2. VHDL OPERATORS	27
Table 3. VHDL CONTROL STATEMENTS	27

LIST OF FIGURES

Figure 1. Scheme for various architectural bodies in one entity	6
Figure 2. Full adder gate structure	8
Figure 3. Full_Adder VHDL structural description	9
Figure 4. Description of and_gate component	9
Figure 5. Description of or_gate component	10
Figure 6. Description of xor_gate component	10
Figure 7. Example of nested blocks	12
Figure 8. D latch block diagram	13
Figure 9. Example of D latch using guarded block	13
Figure 10. Process statement	15
Figure 11. VHDL data type classification scheme	16
Figure 12. Signal assignment example	21
Figure 13. Multiple signal drivers and resolution function	22
Figure 14. Package example	24
Figure 15. VHDL support environment	29
Figure 16. Simulation procedure	31
Figure 17. BLOCK diagram of EPLD design environment	36
Figure 18. EP310 block diagram	37
Figure 19. EP310 macrocell	38
Figure 20. Decomposed EP310 hierarchical block diagram	42
Figure 21. I O primitive signal flow diagram	47
Figure 22. EPLD timing block diagram	52
Figure 23. EP310 hierarchical block diagram with time parameter	53

Figure 24. D register timing diagram	54
Figure 25. Hold time timing diagram	56
Figure 26. EP1800 block diagram	59
Figure 27. Local macrocell	60
Figure 28. Global macrocell	61
Figure 29. EP1800 hierarchical model diagram	63
Figure 30. EP1800 chip overview	64
Figure 31. 16 bit up down counter block diagram	70

ACKNOWLEDGEMENTS

I would like to express my gratitude to my Thesis Adviser, Professor Chin-Hwa Lee, for his advice and assistance in completion of this thesis.

I would also like to thank Professor J.T. Butler and others who contributed their assistance in the accomplishment of this thesis.

Finally, I wish to express my gratitude to my parents whose love and affection has been the force that helped me to achieve this goal of education.

I. INTRODUCTION

A. VHSIC HARDWARE DESCRIPTION LANGUAGE (VHDL)

While system standards such as the Unix and the X Window System continue to be refined and redefined, significant application-level standards are emerging. On one hand, there is explosive growth of electronic computer-aided engineering (CAE) and computer-aided design (CAD). On the other hand, because of the lack of a standard, numerous dissimilar and incompatible CAE/CAD databases have proliferated.

New circuit designs inevitably will adopt part of old designs which were proven to be applicable even with a device technology break-through. Because of the large number of vendor-specific CAE/CAD formats, tools, and languages, the portability of designs are difficult, if not impossible. Consequently, designers tend to redevelop their designs in the new working environment rather than port the existing designs from another incompatible working environment. Different formats and tools continue to grow. It causes system design to be even more inefficient. The need for an industry standard is apparent.

When the Department of Defense launched the Very High Speed Integrated Circuit (VHSIC) project, it was confronted with the problems mentioned above. Furthermore, since most of the government systems tend to have a long service life, the last system on a given program might well be delivered fifteen years after the first production shipment. Both would be expected to remain in operation for at least ten years from that point in time. Technology, on the other hand, is irrevocably non-static. Technology used in the first delivery would be something quite different from those of twenty-five years later. Yet, the operating systems and their maintenance require compatible re-

placement of components. Standardization, if it could accomplish this, could also assist in the procurements from second sources [Ref. 1].

In the early stage of the VHSIC program, an idea arose that these problems might be resolved through the development of a standard hardware description language (HDL). A program was launched in August of 1983 for the development of a VHSIC Hardware Description Language (VHDL) [Ref. 1].

The goal of the program was to develop a language which could simultaneously function as a design automation tool interface and as a mechanism for documenting the design of a transportable electronic system and its components.

In August 1985, version 7.2 of the language was released by the Department of Defense. After the release of version 7.2, the IEEE sponsored the standardization. The goal was the development of an improved standard of the language. The review process was completed by May 1987 and the language reference manual (LRM) was released for industrial review. In December 1987 VHDL was accepted as IEEE-STD-1076-1987.

Initially, the VHDL was geared toward system-level design and documentation. However, after the design of the language has received inputs from many individuals in the computer industry, the VHDL offers not only behavioral constructs [Ref. 2] but also register transfer-level (RTL) and structural (i.e., gate level) constructs. The user thus can describe a circuit at the register and the gate levels as well as the behavioral level.

B. ERASABLE PROGRAMMABLE LOGIC DEVICE (EPLD)

To achieve improved system performance in the marketplace, more and more manufacturers have sought higher levels of integration (functional density) for the electronic components in the design. This led to various forms of custom chips. Yet, the custom chips design has the following problems:

1. Development lead times are relatively long.
2. Design cost are significant.

3. Inventory is dedicated to a specific application which is expensive. This prohibits adequate second sources.
4. Design changes in midstream are not allowed due to lead time and inventory constraints.

The concept of the erasable programmable logic device (EPLD) is to provide the user the benefits of large scale integration circuit without the drawbacks of full custom chips. The benefits of such parts include off-the-shelf availability, minimal design costs, multiple sourcing from distributors, and flexible interchangeable inventory.

The EPLD basically consists of two parts. The first part is the conventional programmable array logic (PAL) structure. The second part is a user-configurable I/O control architecture. Inside the I/O control architecture there are signal path, select switches, and flip-flops.

With improved techniques, the speed and functional density of EPLD's are getting faster and higher. This makes the EPDL even more attractive. It is believed that more special purpose logic designs will use EPLD in place of the custom chips.

C. EPLD MODELING

In this study the VHDL language is used to model two EPLD chips, to find out what kind of problems will be encountered when modeling this type of chips, and to propose the VHDL solutions to these problems.

The best way to describe the EPLD is at a register transfer level. In this study the register transfer level models are used to simulate both the EP310 and the EP1800. These devices are EPLD with approximately 300 and 2100 gates, respectively.

A general introduction of the VHDL language and its working environment is the main subjects of Chapter II. In Chapter III the structure of a small scale EPLD, EP310, is first introduced. It is followed by a general introduction of the EPLD's design environment. Then, the specific problems encountered in modeling the EP310 chip are discussed. Chapter IV introduces the modeling of a large scale EPLD, EP1800. The

advantage and disadvantage of using the VHDL language in modeling a digital circuit design is discussed in Chapter V. At the end, Appendix A contains the VHDL source codes for the EP310 and the EP1800 models. Appendix B contains two simulation application VHDL sources files and their results. Appendix C contains an example of the signal select file used in the VHDL supporting environment and a signal map example which is produced by the VHDL supporting environment. Appendix D contains the procedures of constructing the EP310 model and the EP1800 model in the VHDL 1076 standard supporting environment installed at the Naval Postgraduate School.

II. VHDL FEATURE AND ENVIRONMENT

A. BASIC FEATURES OF VHDL

This chapter constitutes a brief introduction of the VHDL. The advanced features of the VHDL and corresponding examples related to the EPROM modeling will be introduced later when they are encountered.

1. DESIGN ENTITIES

In VHDL, a given section of a logic circuit is represented as a *design entity*. The logic circuit represented can be as complicated as an entire system or as simple as an AND gate. A design entity is defined by an *entity declaration* together with one or more corresponding *architecture bodies* as shown in Figure 1.

An entity declaration basically defines the interface between a given design entity and the environment in which it is used. The actual relationship between these inputs and outputs is specified in the architectural body. This body can specify the behavior of the entity directly such as a primitive body. The architectural body can also be structurally decomposed into simpler components. As shown in Figure 1, "Architecture A" is decomposed into smaller components. Each one of the components is an entity itself, and were binded to the "Architecture A" by "configuration specifications".

Illustrated below is an example that shows the basic features of an entity declaration.

```
entity FULL_ADDER is
    generic(sum_delay,carry_delay: TIME:=25ns);
    port(x,y,cin: in BIT; z,cout: out BIT);
end FULL_ADDER;
```

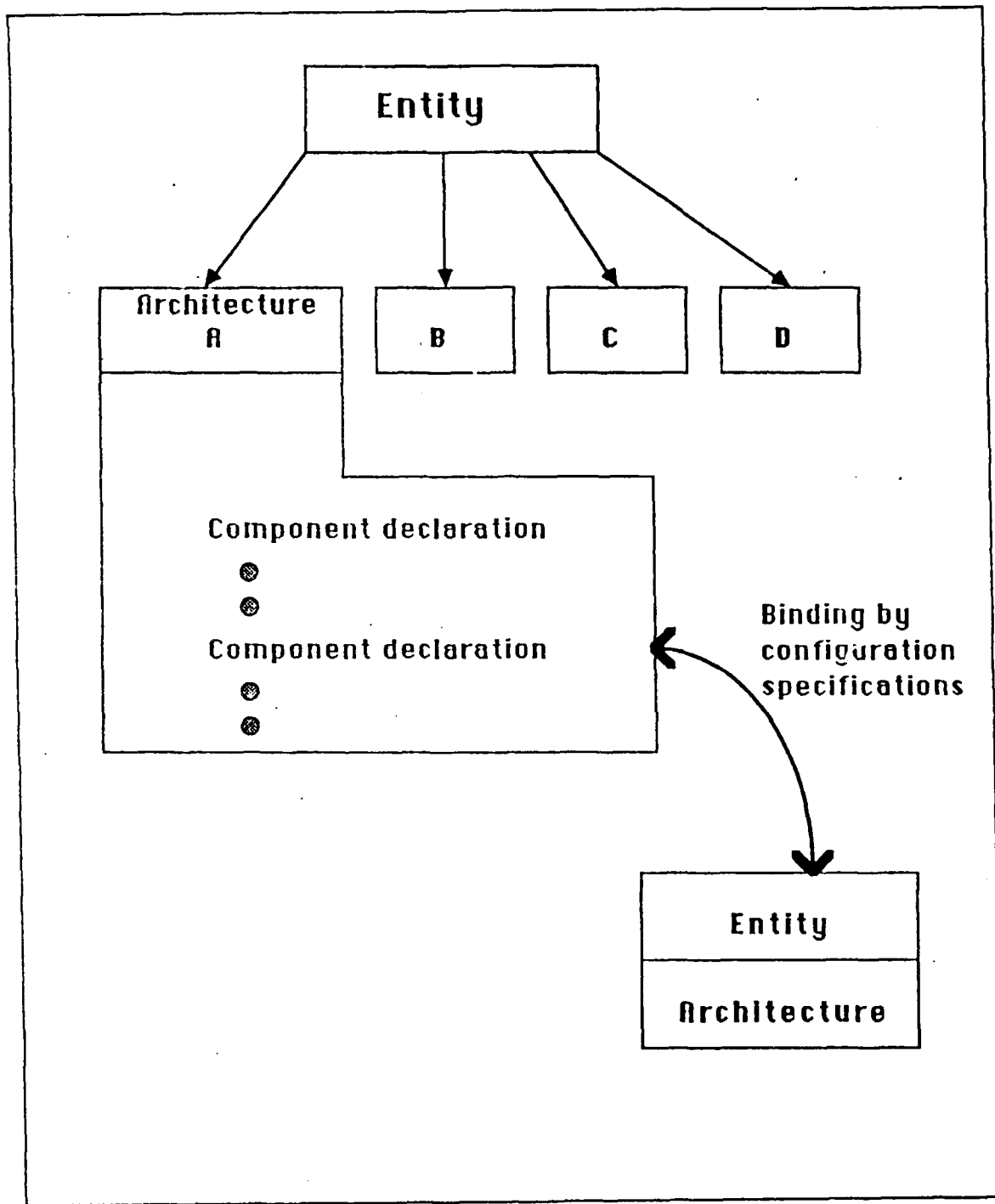


Figure 1. Scheme for various architectural bodies in one entity

The entity name of the above example is "FULL_ADDER" and its ports are "x", "y", "cin", "z", and "cout". Each port has an associated mode, which identifies input, output, and an associated type. In this case, all the signals have the same signal type, i.e., BIT.

The terms *port* and *generic* in the entity declaration above serve as the interface of the module. Port is used to pass the declared signals from external entity into the module. Generic is used to pass the parameters from the external entity into the module. With generic as an interface, it is possible to reuse a design entity whenever practical. For example, generic can be used to pass a faster delay parameter to the module to simulate a faster device. If the variable of a generic appear in a conditional signal assignment statement, then the behavior of the design entity may be changed by changing the value of that generic variable.

Given the entity interface it is also necessary to specify what the "FULL_ADDER" does in an architectural body. An example is shown below:

```
architectural Data_Flow of FULL_ADDER is
begin
    z<=x xor y xor cin after 20ns;
    cout<=(x and y)or(x and cin) or(y and cin) after 25ns;
end Data_Flow;
```

The structural body used in the above example is a behavioral description not involving the actual physical device. In most of the cases, the behavioral description model is the first step in building a real system.

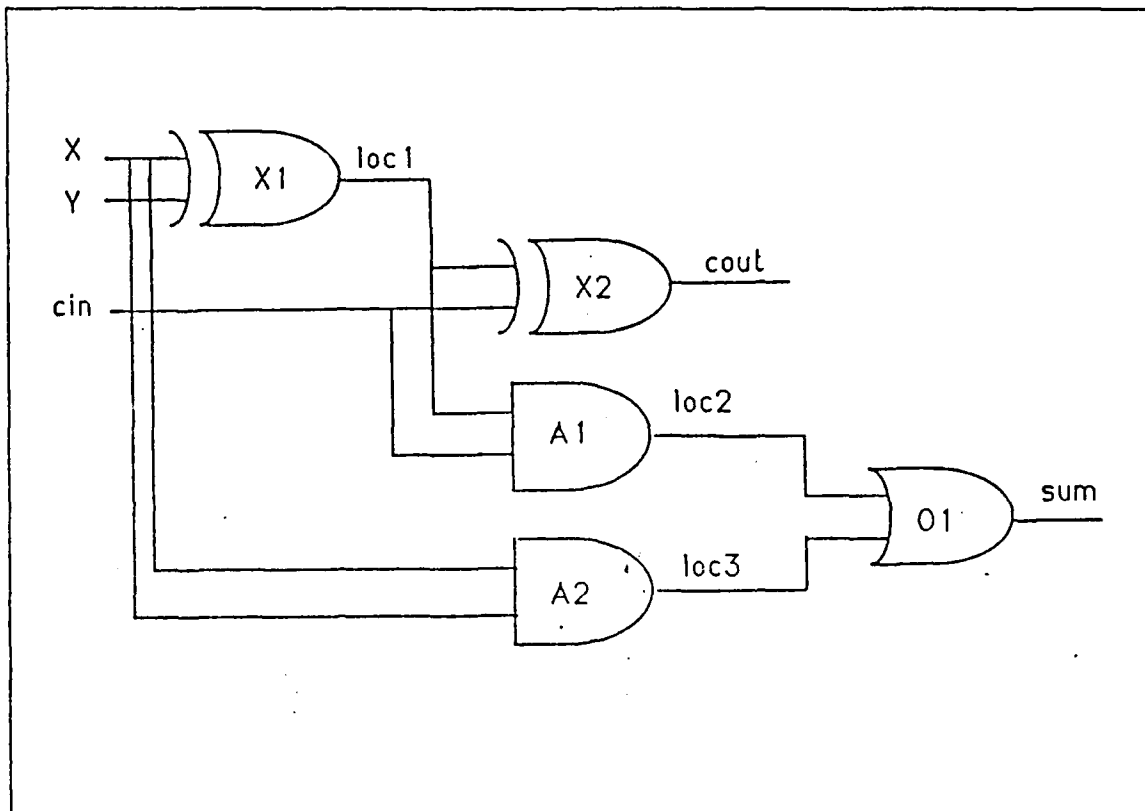


Figure 2. Full adder gate structure

As mentioned before, a single entity may have several architectures corresponding to different ways of realizing the entity. Figure 2 is a physical structure of the "FULL_ADDER", and Figure 3 is the structural realization of it in VHDL.

architectural STRUCTURAL of FULL_ADDER is

```
begin
  component and_gate
    generic (delay: TIME := 5ns);
    port (a,b:in BIT;c out BIT);
  end component;
  component xor_gate
    generic (delay: TIME := 5ns);
    port (a,b:in BIT;c out BIT);
  end component;
  component or_gate
    generic (delay: TIME := 5ns);
    port (a,b:in BIT;c out BIT);
  end component;
  signal loc1, loc2, loc3: BIT;
begin
  X1:xor_gate port map: (x,y,loc1);
  X2:xor_gate port map: (loc1,cin,sum);
  A1:and_gate port map: (cin,loc1,loc2);
  A2:and_gate port map: (x,y,loc3);
  O1:or_gate port map (loc2,loc3, cout);
end STRUCTURAL;
```

Figure 3. Full_Adder VHDL structural description

```
entity and_gate is
  generic (delay: TIME := 5ns);
  port (a,b: in BIT; c: out BIT);
end

architecture BEHAVIOR of and_gate is
begin
  c <= a and b;
end BEHAVIOR;
```

Figure 4. Description of and_gate component

```

entity or_gate is
    generic (delay: TIME := 5ns);
    port (a,b: in BIT; c: out BIT);
end

architecture BEHAVIOR of or_gate is
begin
    c <= a or b;
end BEHAVIOR;

```

Figure 5. Description of or_gate component

```

entity xor_gate is
    generic (delay: TIME := 5ns);
    port (a,b: in BIT; c: out BIT);
end

architecture BEHAVIOR of xor_gate is
begin
    c <= a xor b;
end BEHAVIOR;

```

Figure 6. Description of xor_gate component

Three components, xor_gate, and_gate, and or_gate, are declared in the declaration section of the architecture "STRUCTURAL" in Figure 3. The port specification used in the components of the "FULL_ADDER" indicates that there are two input signal "a", "b", and an output signal "c". The entity of these components have the identical port specifications as the "FULL_ADDER" does. (see Figure 4, Figure 5, and

Figure 6). Three signals loc1, loc2, and loc3 are also declared in the "FULL_ADDER". These declared signals are local signals, which are visible only inside the "FULL_ADDER" entity.

In the "FULL_ADDER" entity, five components are *instantiated* after the key word **begin**; that is, five specific instances of general component entities are created. In each instantiation, there is a unique label associated with it (i.e., X1, X2, A1, A2, and O1) as well as a *port map*. The *generic map* is optional. If the generic map is not stated in the instantiation, the assigned default value will be used. In the above case, since all the component instantiations does not state their generic map, the generics of the components will use their default value, i.e., five nanosecond delay.

The port map creates an association between the inputs/outputs of the component declaration and the signals in the instantiated components. In the "FULL_ADDER" case the association is "by position". *Named association* can also be used, which looks like the following:

```
A1: and_gate port map (c=>loc2, a=>cin, b=>cin);
```

2. CONCURRENT STATEMENTS

Some of the most important characteristics of the concurrent statements are described in this section. Concurrent statements are used to define interconnected blocks and processes that jointly describe the overall behavior or structure of a design entity. Concurrent statements are executed asynchronously with respect to each other.

a. BLOCK STATEMENT

The basic concurrent statement in VHDL is the block statement. The block consists of the declaration section as well as an executable section. The executable section can have all possible concurrent statements, which may include other block statements. The architectural body of the entity itself is basically a block. Consider the example shown in Figure 7. Here, the architectural body "BLOCK_STRUCTURE" can

be seen as an outer block. The block "A" under the "BLOCK_STRUCTURE" is its inner block. In general, any number of nestings of blocks are possible. Note that in Figure 7 the text following the symbol "--" are comments in VHDL and will not be executed.

```

architecture BLOCK_STRUCTURE of FULL_ADDER is
  -- Outer Block Declaration Section
  signal loc1 : BIT;
begin
  -- Outer Block Executable Section
  loc1 <= x xor y after delay;
  sum <= loc1 xor cin after delay;
  A:block
  -- Inner Block A Declaration Section
    signal loc1,loc2,loc3,loc4 : BIT;
  begin
    -- Inner Block A Executable Section
    loc1 <= x and y after delay;
    loc2 <= y and cin after delay;
    loc3 <= loc1 or loc2 after delay;
    loc4 <= loc3 or loc4 after delay;
    cout <= loc3 or loc4 after delay;
  end block A;
end BLOCK_STRUCTURE;

```

Figure 7. Example of nested blocks

In VHDL, the block statement contains an optional guard expression. If there is an expression immediately following the key word **block**, then an implicit signal called *guard* with a type **BOOLEAN** will be created. The value of guard is dependent on the condition of the expression that follows the block. When a "guard condition" becomes **TRUE**, it can enable certain types of statements inside the block. By using a "guarded block" it is possible to model the operation of a synchronous circuit.

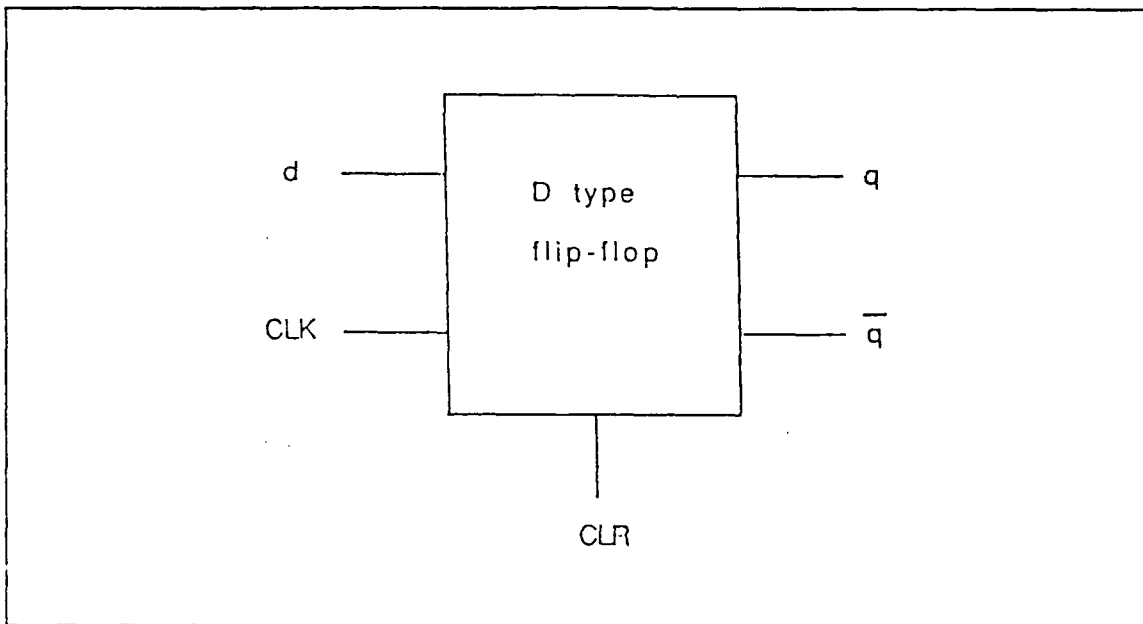


Figure 8. D latch block diagram

```

entity D_Latch is
  port(CLK,CLR,D:in BIT; Q,Q_not:out BIT);
end D_Latch;

architecture GUARDED_CLOCK of D_Latch is
begin
  block(CLK = '1' or CLR = '1')
    signal S: BIT;
  begin
    S <= guarded '0' when CLR = '1' else
      D;
    Q <= S after 10ns;
    Q_not <= not s after 10ns;
  end block;
end GUARDED_BLOCK;
  
```

Figure 9. Example of D latch using guarded block

A "guarded block" example is given by modeling the D latch as shown in Figure 8. This D latch will reflect the value of the input signal "d" when the clock (CLK) is high, i.e., equal to '1'. The latch also contains an asynchronous clear (CLR)

input. When $CLR = 1$, the latch is reset. The asynchronous clear overrides the clock. The corresponding latch model is shown in Figure 9. The block header expression "(CLK = '1' or CLR = '1')" define the value of the implicit guard. The statements contains the key word **guarded** will not be executed unless the value of guard is TRUE. In the D latch case, if the guard is FALSE, the local signal S will remain the same value. If the guard is TRUE and the CLR is a '1', then S will be assigned to '0'; otherwise S will follow the input signal D. Note that the D latch illustrated above is a level-sensitive device. It will be shown in Chapter III that the guard is not restricted to level-sensitive. Edge-sensitive guard is available as well.

b. PROCESS STATEMENT

Another major modeling element in VHDL is the *process*. A process statement defines an independent sequential process. The process can represent the behavior of some portion of a design. An example is shown in Figure 10. Note that the process begin with the key word **process**(line). The parameter inside the parenthesis is called the sensitivity list. Whenever the signal inside the sensitivity list change, the process is activated and the statements within the process are executed.

3. DATA TYPES

The *type* is an important feature of VHDL. The VHDL is strongly typed, which means that inadvertent mixing of types in an operation will be flagged as an error. The strong typing features are very helpful for capturing the designer's intent; they also help to check the design data-flow correctness.

A type is characterized by a set of values and a set of operations. A *subtype* is a subset of the values of a type. The set of operations defined for a subtype include the operations defined for the parent type; however, the assignment operation to an object having a given subtype only assigns values that belong to the subtype.

```

process (line)
begin
  pulse <= '1' after 5ns, '0' after 15ns;
end process;

```

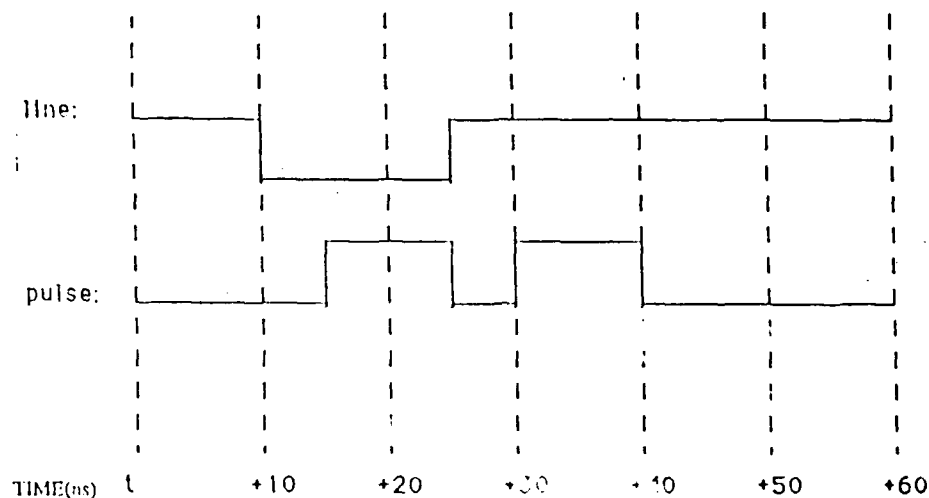


Figure 10. Process statement: (Pulse Converter)

In the VHDL the set of all integers are predefined as type INTEGER, and all positive integers are predefined as subtype POSITIVE in the following declaration

```

subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

```

Where the attribute 'HIGH will yield the upper bound value of the INTEGER. Here, with the key word range the subtype POSITIVE will have the same upper bound value as the INTEGER has. According to the subtype operation rule mentioned above, the subtraction operation implied for INTEGER, may not be used in subtype POSITIVE, because the subtraction of two positive numbers may produce a negative value.

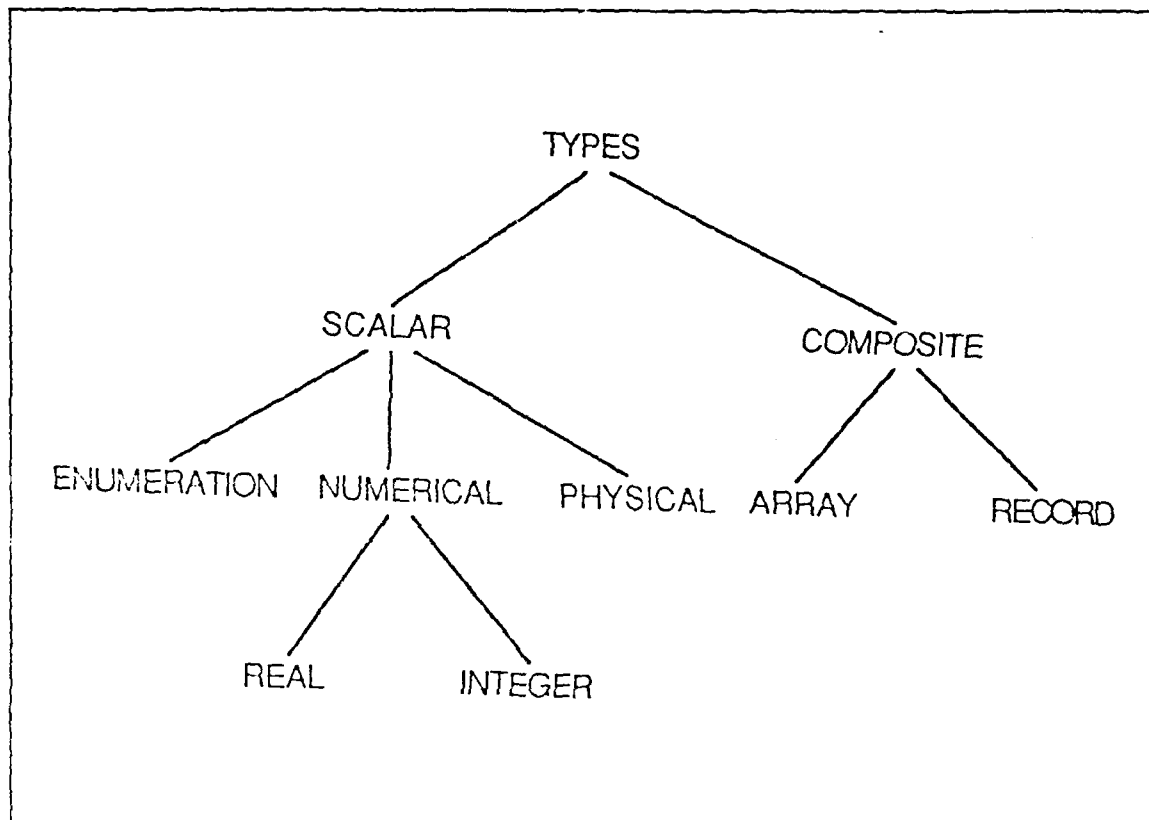


Figure 11. VHDL data type classification scheme: [From Ref. 3]

The "VHDL Language Reference Manual" gives a useful data type classification scheme which is illustrated in Figure 11. Data types are classified as *scalar* (one-dimensional) or *composite* (multidimensional). The *enumeration* type is the most frequently used scalar types. For example, the predefined type BIT is an enumeration type, which is defined as follows:

```
type BIT is ('0','1');
```

Numeric types are either INTEGER or REAL. The REAL type can be very useful in modeling an analog-to-digital interface or a signal processing algorithm. Some type declaration examples are illustrated below:

```
type INDEX is range 0 to 9; --integer type.
```

```
type VOLTAGE is range 0.0 to 10.0; --real type.
```

Each of the type declared in above example has a range. The base type (i.e., either INTEGER or REAL) is implied by the values in the range.

The VHDL also provides the *physical* type, which is used to represent some physical quantities (such as, time, voltage, and capacity). TIME is a predefined physical type in VHDL, and its declaration is shown below:

```
type TIME is range implementation_defined
  units
    fs;                -- femtosecond
    ps  = 1000fs;      -- picosecond
    ns  = 1000ps;      -- nanosecond
    us  = 1000ns;      -- microsecond
    ms  = 1000us;      -- millisecond
    sec = 1000ms;      -- second
    min = 60sec;       -- minute
    hr  = 60min;       -- hour
  end units;
```

Note that the base unit in TIME is femtoseconds (fs). Both the range and the base unit are user selectable. But, the upper bound of the range cannot exceed the host machine limit. With the range from 0 to 1E20, the TIME can represent up to 27.7 hours (100,000 second).

The *Composite* types are either *array* types or *record* types. Below is an array type declaration example:

```
type BYTE is array(0 to 7) of BIT;
```

As declared above, BYTE is an array type of eight array elements. BIT is the base type of these array elements. *Index range* of the BYTE is constrained and is from "0 to 7". Another example with unconstrained index range is shown below:

```
type BIT_VECTOR is array( NATURAL range <>) of BIT;
```

The expression "NATURAL" in BIT_VECTOR above means that the index has a subtype of natural number, i.e., non negative number. The symbol "< >" following the key word **range** stands for an undefined range. The user must specify this range when it is employed. For example, BIT_VECTOR(0 to 7) for ascending range, or BIT_VECTOR(7 downto 0) for descending range.

As the name implies, a record type is a composite type consisting of a number of fields. For example, type DATE could be defined as a record type as follows:

```
type DATE is
  record
    DAY : INTEGER range 1 to 31;
    MONTH : MONTH_NAME;
    YEAR : INTEGER range 0 to 3000;
  end record;
```

The type MONTH_NAME would be an enumeration type consisting of the names of the months.

As mentioned before, the variable in VHDL are strongly typed, and this means that the objects with different types cannot be involved in the expression directly. An example is illustrated as follows:

```

-- declaration part
signal A : BYTE := "00001111";
signal B : BIT_VECTOR(0 to 7) := "11110000";
signal C : BYTE;
-- expression part
C <= A and B;    -- this expression will NOT be accepted by VHDL

```

Although the A and the B all have eight bits, because they are different types, the VHDL will not accept the expression shown in the above example. Note that in the declaration part, the signal A is assigned with an initial value via the symbol ":= " immediately following the type.

4. CLASS OF OBJECTS

In VHDL there are three classes of objects: *constant*, *signals*, and *variables*.

a. Constants

A constant is an object whose value may not be changed. Some examples of constant declarations are:

```

constant PI : REAL :=3.1416;
constant MESSAGE : STRING(1 to 13) := "demonstration";

```

Note that each constant declaration must includes the name, the type, and the value of the constant.

b. Variables

Variables are objects whose values can be changed. When a variable is created by a declaration, a container for the object is created along with it. Variables are changed by executing a variable assignment statement: for example,

```

A := B + C;

```

Variable assignment statements have no time dimension associated with them, i.e., their effect is felt immediately. Thus, variables have no direct hardware correspondence. But, they are useful in algorithmic representations. Some examples of the variable declarations are:

```
variable FLAG : BOOLEAN := TRUE;
```

```
variable COUNT : INTEGER := 0;
```

The variable declarations specify the name, type, and optionally an initialization value for the variable. Variables used in a process block are considered to be *static*; that is, the value of the variable is maintained by the simulator until it is changed by a variable assignment statement.

c. Signals

Signals are objects whose values may be changed, and the execution has a time dimension. Signal values are changed by signal assignment statements. The signal assignment statements are evaluated whenever one of the right hand elements of the statements changed its value. An example is shown in Figure 12. Note that the signal assignment statement uses the " \leq " symbol in order to differentiate it from the variable assignment statement. The "after 10ns" in Figure 12 means that the "local" will take on its new value 10 ns later from the present simulation time. If the signal assignment statement has no **after** clause, then it is equivalent to "after 0ns".

All the signal assignment statements in the same process level are concurrently executed in the same simulation cycle. Consequently, the positions of the signal assignment statements in the same process level are irrelevant. Thus, the signal assignment statements in Figure 12 can also be rewritten as

```
d <= local and c;
```

```
local <= a and b after 10 ns;
```

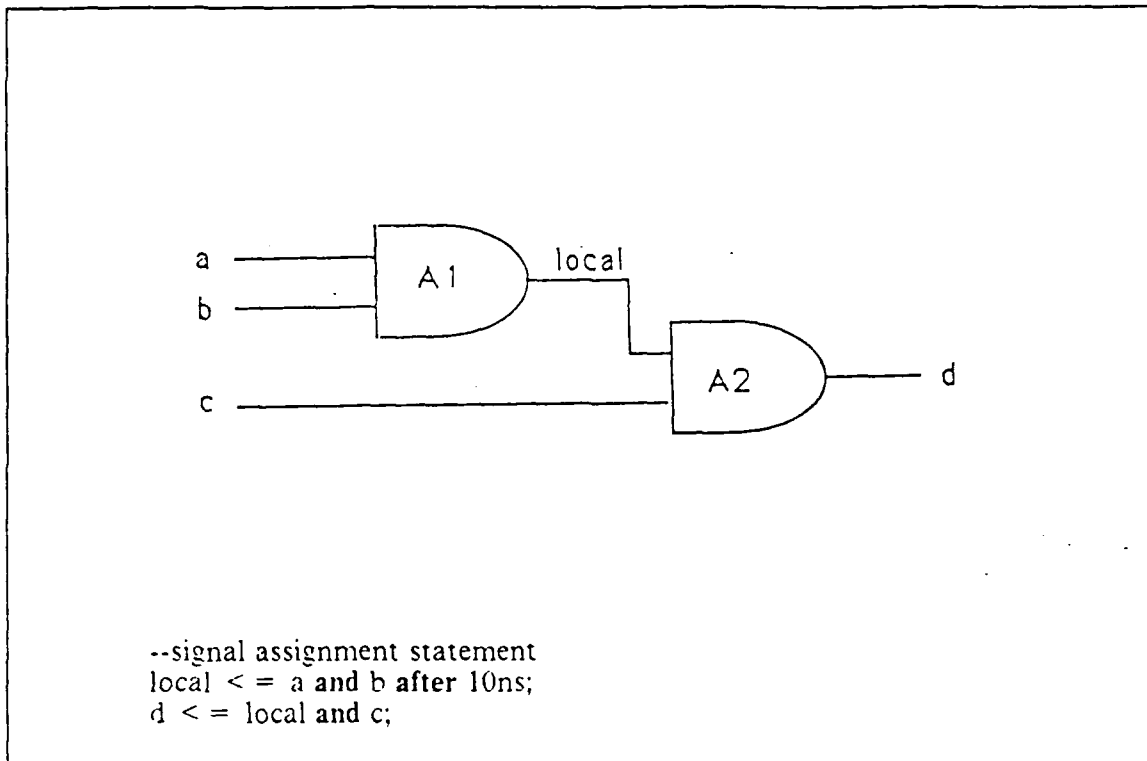


Figure 12. Signal assignment example

One of the difference between variables and signals is that signals can have multiple queuing containers called *drivers*. The value of the signal is a function of all the related drivers. An example related to multiple drivers is shown in Figure .13. In the figure two signal assignment statements A and B assign values to the same signal X. For each signal assignment, a driver is created to hold the result of that assignment. In this example, drivers are labeled Dax and Dbx. The value of the signal X is determined by a *resolution function* F in this example. The resolution function F is user defined. It is activated whenever the drivers receive new values. The value of the signal is updated to the new value that coming from the resolution function. In Chapter III, the use of a resolution function to model the EP310 will be shown.

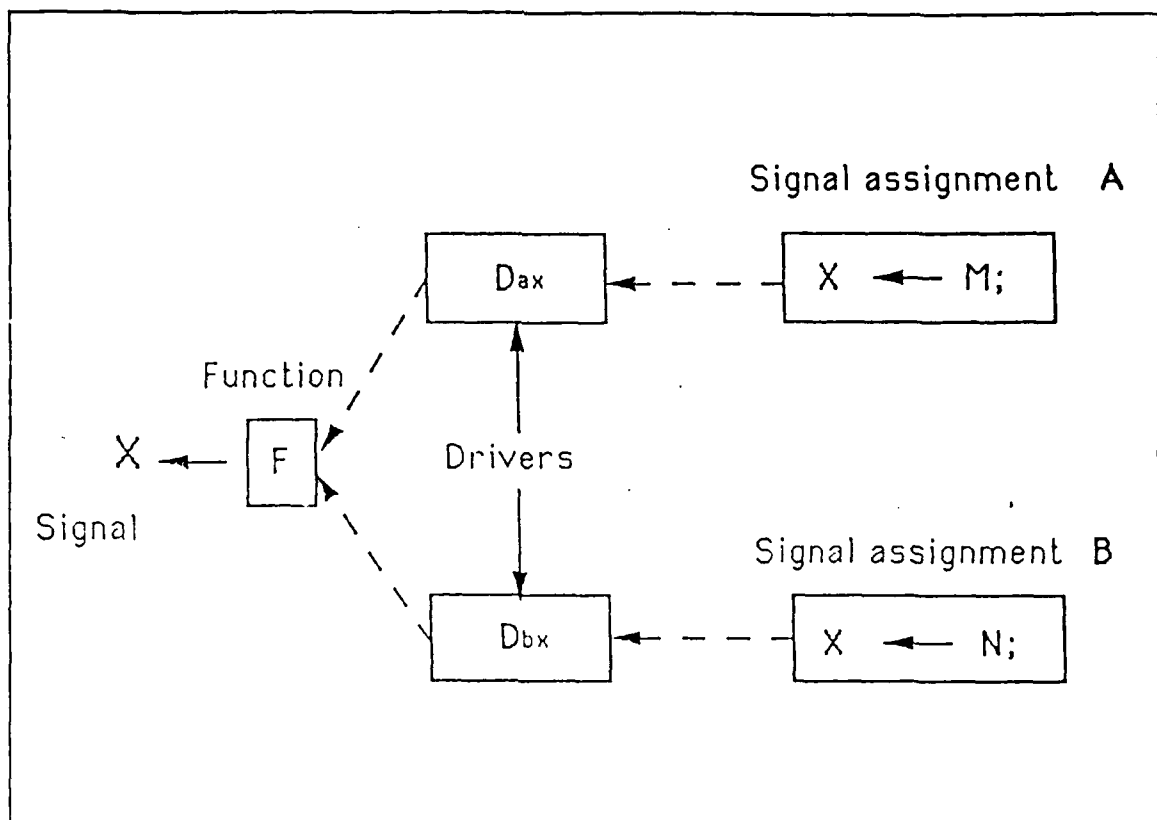


Figure 13. Multiple signal drivers and resolution function: [From Ref. 3]

5. ATTRIBUTES

An *attribute* defines and evaluates some characteristics of a named object. Some attributes are predefined and are related to types, ranges, values, signals, or functions [Ref.4: pp.4-14,15]. Signal attributes are particularly important in modeling. Some predefined signal attributes are shown in the following examples:

1. **S'EVENT** returns a **BOOLEAN** value. If an event has occurred on **S** during the current simulation cycle the returned value is **TRUE**; otherwise, it returns the value **FALSE**.
2. **S'STABLE(T)** is of type **BOOLEAN**. It is **TRUE** if **S** has been stable for the last **T** time units. If **T** is zero, it is written as **S'STABLE**.
3. **S'DELAYED(T)** is the value of **S**, **T** time units earlier. It has the same type as **S**.

These attributes are useful in detecting signal changes and will be used in the later chapters.

Another useful set of attributes are those associated with arrays. For example, suppose that an array variable was defined as follows:

```
variable A : BIT_VECTOR (0 to 15);
```

Table 1. ATTRIBUTE VALUES

Attribute	Value
A' RANGE	0 to 15
A' LENGTH	16
A' LEFT	0
A' RIGHT	15

Then, the set of attributes of variable A would have values indicated in Table 1. Many other useful attributes are defined in the "VHDL Language Reference Manual." [Ref. 4: pp.14-1.9]

6. SUBPROGRAM AND PACKAGES

Subprograms define algorithms for computing values or exhibiting certain behavior. There are two forms of subprogram: *procedures* and *functions*. A procedure call is a statement; a function call is an expression that returns a value. The definition of a subprogram can be given in two parts:

1. A subprogram declaration defining its calling conventions.
2. A subprogram body defining its execution.

Packages, like subprogram, may be defined in two parts. A *package declaration* defines the visible contents of a package; a *package body* provides the hidden details. In particular, a package body contains the bodies of all subprograms declared in the package declaration.

```

package Multiplication is
  type BIT_ARRAY is (INTEGER range < >) of BIT;
  function BitArray_Int (bits: BIT_ARRAY) return NATURAL;
  function Int_to_BitVec (int, length: NATURAL) return BIT_ARRAY;
  procedure Obtain_Product (a,b: in BIT_ARRAY;
                           c: out BIT_ARRAY;
                           ov: out BIT);
  -----other declarations
  -----
end Multiplication;

package body Multiplication is
  function BitArray_to_Int (bits: BIT_ARRAY) return NATURAL is
    variable result: NATURAL := '0';
  begin
    for i in bits'RANGE loop
      result := result * 2;
      if bits(i) = '1' then
        result := result + 1;
      end if;
    end loop;
    return result;
  end BitArray_to_Int;

  function Int_to_BitArray -----
  -----
  -----

  procedure Obtain_Product( a,b: in BIT_ARRAY;
                           c: out BIT_ARRAY;
                           ov: out BIT) is
    variable local: INTEGER;
    constant limit: INTEGER := 2 ** a'LENGTH;
  begin
    local := BitArray_to_Int(a) * BitArray_to_Int(b);
    if local > limit-1 then
      local := local mod limit;
      ov := '1';
    else
      ov := '0';
    end if;
    c := Int_to_BitArray(local, a'LENGTH);
  end Obtain_Product;

end Multiplication;

```

Figure 14. Package example

Shown in Figure 14 is a package consisting of a type declaration and the interfaces for two functions and a procedure. The code for the functions and the procedure are given in the package body. If a package contains no subprograms, a package body is not required. The package can be accessed by placing a **use** clause before the interface description of an entity. For example, if there is an entity called **MULTIPLIER8** need to use the procedure **Obtain_Product** inside the **Multiplication** package in Figure 14. One could do this as follows:

```

use WORK.Multiplication

entity MULTIPLIER8 is
    port (a,b: in BIT_ARRAY(0 to 7);
          prod: out BIT_ARRAY (0 to 7);
          ov: out BIT);
end MULTIPLIER8;

architecture BEHAVIORAL of MULTIPLIER8 is
begin
    process (a,b)
        variable c: Multiplication.BIT_ARRAY (a'RANGE);
        variable loc: BIT;
    begin
        Multiplication.Obtain_Product (a,b,c,loc);
        prod <= c after 100ns;
        ov <= loc after 100ns;
    end process;
end BEHAVIORAL;

```

Note that when the package feature is referred to by the statement inside entity "Multiplier8", the package name must be placed in front of the package feature in order to establish "visibility" between those referred items and the entity. For example, the procedure "Obtain_Product" used in the entity "MULTIPLIER8" must have the package name "Multiplication" in front of it. If the clause `.all` is added at the end of the `use` statement, i.e., `use WORK.Multiplication.all`, then all the declarative items as well as any of its subprogram bodies contained in Multiplication package are "visible" within the entity MULTIPLIER8. In this way, there is no need to repeat the package name in front of the referred package items.

Packages are very useful language features. Design groups can use group standard packages that contain the type declarations and subprograms related to their projects. As it will be seen in modeling the EPLD, a group standard package EPROM will be built and extensively used in the program.

The VHDL language defines a package STANDARD that can be used by all entities. This package contains the definitions for types BIT, BIT_VECTOR, BOOLEAN, INTEGER, REAL, CHARACTER, STRING, TIME, as well as subtype POSITIVE and NATURAL.

7. OPERATORS AND CONTROL STATEMENTS

The operators and control statements of VHDL are similar to those of the other high level languages, especially Ada. A complete set of operators are shown in Table 2.

Table 2. VHDL OPERATORS

Class	Class Members
Logical	not and or nand nor xor
Relational	= /= < <= > >=
Adding	+ - &
Signing	+ -
Multiplying	 / mode rem
Miscellaneous	* abs

Table 3. VHDL CONTROL STATEMENTS

IF	LOOP	RETURN
CASE	NEXT	WAIT
	EXIT	

The control statements of VHDL are shown in Table 3. Note that most of the statements are very general, and only reserved word **wait** need a further discussion. The wait statement causes the suspension of a process or a procedure. In the real physical system, a process will frequently pause in its execution while waiting for a event to occur or a time period to elapse. Once the awaited event has occurred or the time period has elapsed, execution of the process resumes.

In VHDL the **wait** syntax is shown as follows:

wait on sensitivity_list until condition for time_out

The statement suspends the process until a signal in the "sensitivity_ list" changes, at which time the "condition" clause is evaluated. The "condition" clause is an expression of type BOOLEAN. If it is TRUE, the process resumes. The "time_out" clause sets the maximum wait time after which the process will resume. As an example:

`wait on x, y until (Z=0) for 100 ns;`

This statement will suspend a process until either "x" or "y" changes, then the expression "z = 0" is evaluated, and if the value is TRUE, the process will resume. The process will resume after 100 ns, even if the signals do not change or the condition is FALSE.

Two major applications of the `wait` construct in modeling are the modeling of component interaction and oscillator behavior [Ref. 3: pp. 55]. The `wait` statement gives the designer additional freedom in writing high-level behavioral models.

B. VHDL SUPPORTING ENVIRONMENT

The purpose of having a VHDL supporting environment is to assist hardware designers in making efficient use of the capabilities of the VHDL language. A typical VHDL supporting environment includes the design library, the Analyzer, and the Simulator [Ref. 1]. Currently, there is only one VHDL supporting environment (Intermetrics Standard IEEE 1076 VHDL Supporting Environment) installed at the Naval Postgraduate School. The following discussion will be based on this VHDL supporting environment.

The Intermetrics Standard IEEE 1076 VHDL Supporting Environment consists of a *Design Database* and four software components as shown in Figure 15. A general descriptions of this VHDL supporting environment are listed as follows:

1. *Design Database.* The Design Database is the central part of the system. The database of the hardware descriptions and related information are all stored in here.
2. *Analyzer.* The Analyzer checks the hardware descriptions for syntactic and static semantic correctness. It also translates VHDL text to the Intermediate VHDL Attributed Notation (IVAN) form, and installs the translation into the Design Database.
3. *Simulator.* The Simulator computes the behavior of a hardware model described in VHDL and thus provides a mean of checking dynamic semantic correctness. The Simulator constructs simulatable modules from IVAN data, executes these modules and generates reports on the runs. The Simulator consists of five sub-components [Ref. 5], four of them are related to user and will be discussed as follows:

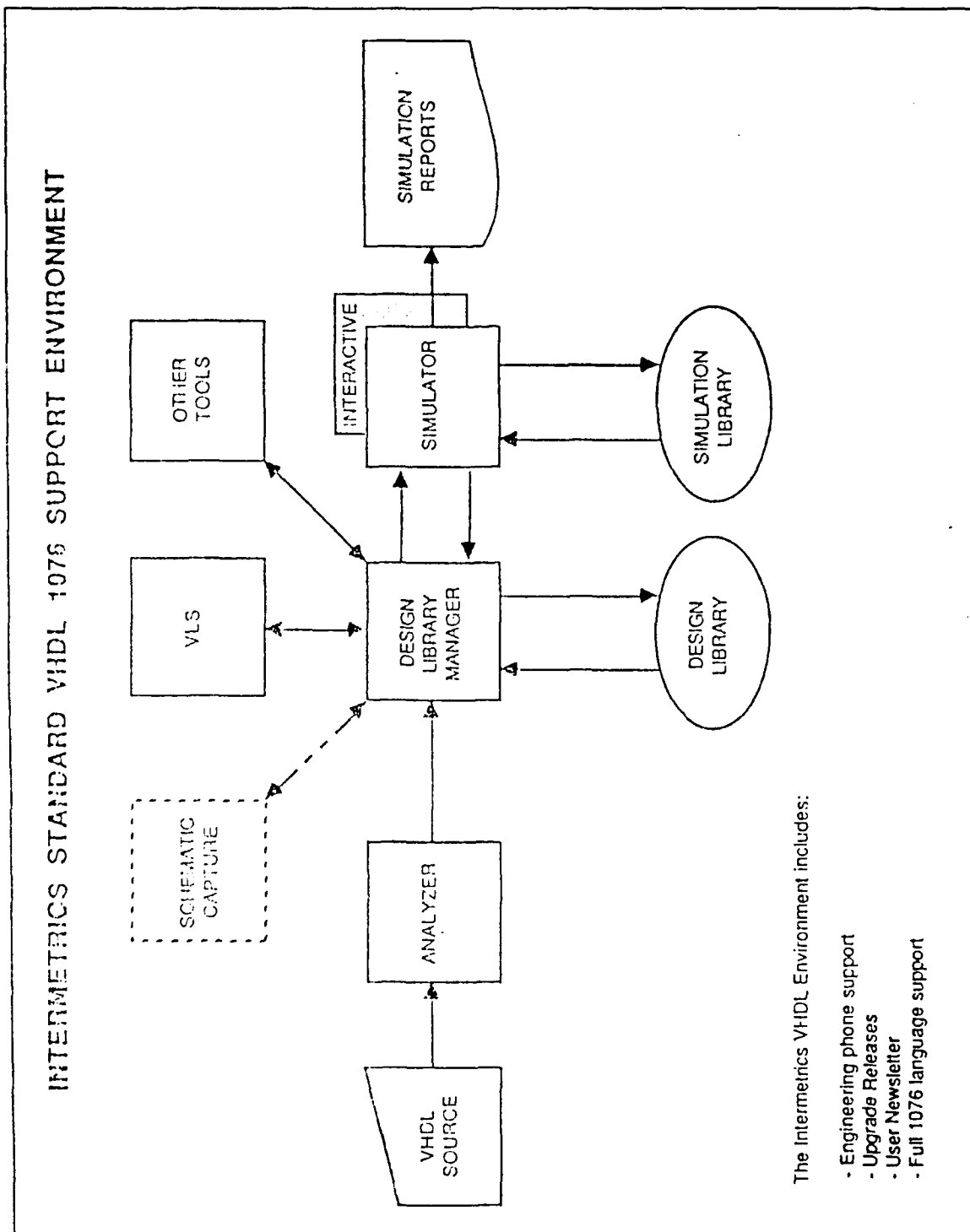


Figure 15. VHDL support environment: (courtesy of Intermetrics Inc.)

- a. Model Generator (MG). The Model Generator translates IVAN data into C source code, and compiles it to the simulation module.
- b. Build. Build links separately model-generated units and makes an executable *Kernel* in the Design Database.
- c. Sim. Sim invokes the Kernel and passes user-defined runtime parameters to the model. Execution of the simulation model will typically result in the production of a file containing signal history.
- d. Report Generator (RG). The Report Generator produces human readable reports from the file of signal histories. The selection of signals and the format of the report are determined by report control language file supplied by the user.
- e. VHDL Library System (VLS). VLS provides the commands necessary for the Design Database.
- f. Design Library Manager. The Design Library Manager is the database management system used by the Analyzer, Simulator, and VLS to access data in the Design Database.

1. SIMULATION PROCEDURE

After a VHDL model has been created, the model needs to be simulated. The simulation procedure is shown in Figure 16. The first step, as shown in Figure 16, is to invoke the Analyzer by typing the key word **VHDL** followed by the program file name as shown in the file Batch of Appendix D. If the model has no error, an IVAN form data will be created. Two rules govern how VHDL design units are analyzed:

1. A body cannot be analyzed before its interface.
2. A unit that references a package cannot be analyzed before that package.

After the VHDL program has been translated to the IVAN form, it then can be sent to the Model Generator by typing **MG** preceding the design VHDL unit name, i.e., the name of the entity or the package just been analyzed. If the VHDL unit is a package body, the qualifier *body* must follow the **MG** key word.

Each model when simulated needs to have a *top entity*. This top entity may be thought of as a unit that contains a model under test. The top entity itself cannot have any ports, but, it can have generics for passing parameters from external entity or VHDL supporting environment. An example is shown in the TEST_BENCH of Ap-

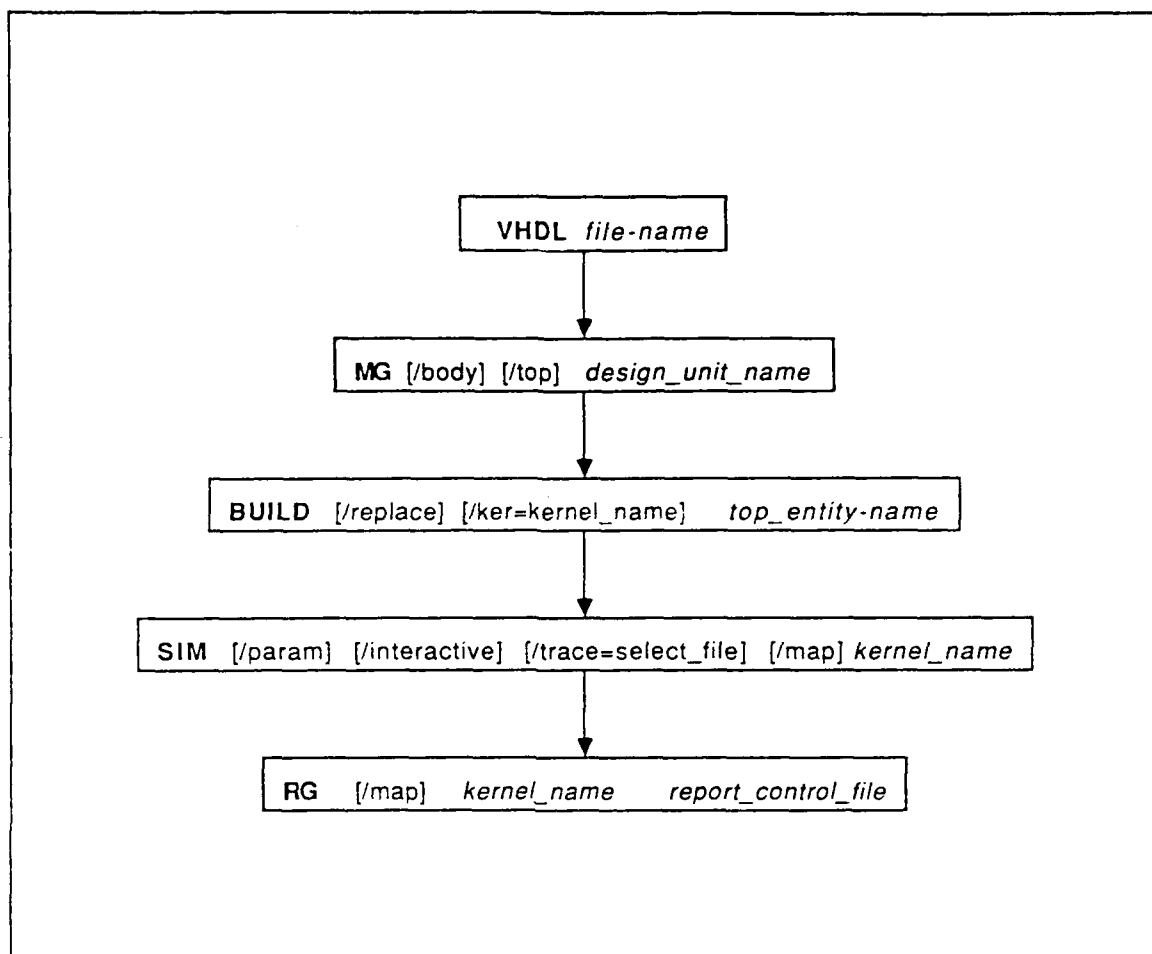


Figure 16. Simulation procedure

pendix B. In the structure of the top entity, the user must provide a test vector mechanism in order to conduct the necessary testing. When generating this top entity, a qualifier "top" must follow the key word **MG**.

After all the object modules have been "Model Generated", they can be "linked" by typing **BUILD**. This will produce an executable Kernel in the *work* library. The user can assign a kernel name by using the qualifier "ker= name" following the key word **BUILD**.

The **SIM** key word will execute the Kernel and generate a *Run* in the working library. The **SIM** can be followed by a qualifier `"/trace = select"`, where "select" may be any file name which contains the pathnames of the selected signals. Consequently, the Simulator will record the histories of these selected signals. In this way, the user can discard those irrelevant internal signals, and use the limited memory to keep the wanted signal information. If no trace qualifier is specified, which is the default, then the histories of all the signals will be kept.

As mentioned above, the top entity can only have the generic declaration, which can receive the values provided by the **SIM** qualifier `"/param = "`. With this feature, the user can control the simulation time or even change the operation of the model.

Besides the non-interrupting simulation mode, the VHDL supporting environment also provides an interactive simulation mode as shown in Figure 15. In this mode, the user can set the breakpoint, see the signal transitions, and change the status of the signals. The interactive simulation mode is a very useful tool in debugging.

2. REPORT GENERATOR

The Report Generator will produce a readable report file from recorded information. Its output signal and format are controlled by the report control file. An example file is shown in Appendix B. The command to generate the report is as follows:

```
RG control_file report_output_file
```

Note that only those signals with histories preserved in the simulation can be generate in the report.

The qualifier `"/map"` can be used with key word **SIM** and **RG**, this qualifier can produce a signal map, which contains all the signals used in the model. With this signal map, the user can easily find out the wanted signal *pathnames*. An signal map example is shown in Appendix C.

3. VHDL LIBRARY SYSTEM (VLS)

The VLS allows user to interact with the VHDL Design Database. The user can create his own library. A library is either a *primary* library or a *secondary* library. A primary library may contains the following kinds of data:

- VHDL units, i.e., entity declarations, architecture bodies, package declarations, package bodies, configurations.
- Simulation Kernel.
- Simulation runs.
- Other libraries.

The secondary library may contain only package bodies, architecture bodies, and other secondary libraries. The purpose of secondary library is to allow users to experiment with their designs by using alternative package bodies or architecture bodies.

The VHDL Library System is entered by typing the **VLS** key word. After entering the VLS, the user can make his own library by typing the key word **MAKELIB** followed by a physical library name, an example is shown below:

```
MAKELIB <<user_account.EPROM>>
```

Note that the symbol "< < > >" means that the name inside is the physical name of the created library. The user_account is the user login name. In VLS, this name is treated as a *root* library name. Every user must have a root library. In the example above, a library called EPROM was created under the root library.

There are two predefined libraries: STD and WORK. STD is the *logical* name of the standard library. WORK is the logical name of the current user working library. The logical name can be declared as follows,

```
DEFINE EPROM <<user_account.EPROM>>
```

All the created VHDL units will be stored in the current working library whose default is the root library. If the user wants to change the current working library, he

or she has to type the key word **SETLIB** followed by the physical or logical library name in the VLS environment. For example,

```
SETLIB EPROM
```

will change the working library to < <user_account.EPROM> > .

The design group members may share each others libraries. For example, assume that there are two users, "phred" and "janus". If "janus" creates a entity that uses the package EP310_PACK in the library EPROM, and the EPROM belongs to "phred", she may declare

```
library EPROM;

use EPROM.EP310_PACK.all;

entity-----
-----
-----
```

With the shared libraries, the user can avoid the redundant components and standardize the design.

III. MODELING THE EP310

A. INTRODUCTION OF THE EP310

The EP310 is an Erasable Programmable Logic Device (EPLD) manufactured by the Altera corporation. A user can use the CAE design tools, as shown in Figure 17, to configure the connections in the programmable AND logical array and the flexible output feedback section of an EP310. As shown in Figure 17, the user can do his design via schematic capture entry, state machine entry, netlist entry, or boolean equation entry. Once the design was finished, it can be processed, and a JEDEC file is produced. The JEDEC file is a file with the standard data transfer format from the design system to the hardware programmer unit. An EPLD can be physically configured by a "hardware programmer" with the JEDEC file as input.

Externally, the EP310 provides 10 dedicated inputs. One of which may be used as a synchronous clock input. Eight I/O pins, shown in Figure 18, may be configured for input, output or bi-directional operation.

Figure 18 shows the complete EP310 block diagram, and Figure 19 shows the basic EP310 macrocell. The internal architecture of a microcell is organized in a sum of products (AND-OR) structure. Inputs to the programmable AND array, shown running vertically in Figure 19, come from two sources:

1. The true and complement of the 10 dedicated input pins.
2. The true and complement of 8 feedback signals, each one originating from a I/O Architecture Control Block.

The 36 input AND array, as shown running horizontally in Figure 19, is called the product term. There are 8 identical macrocells in an EP310. Each macrocell has 9 product terms. Therefore, the total number of product term is 72.

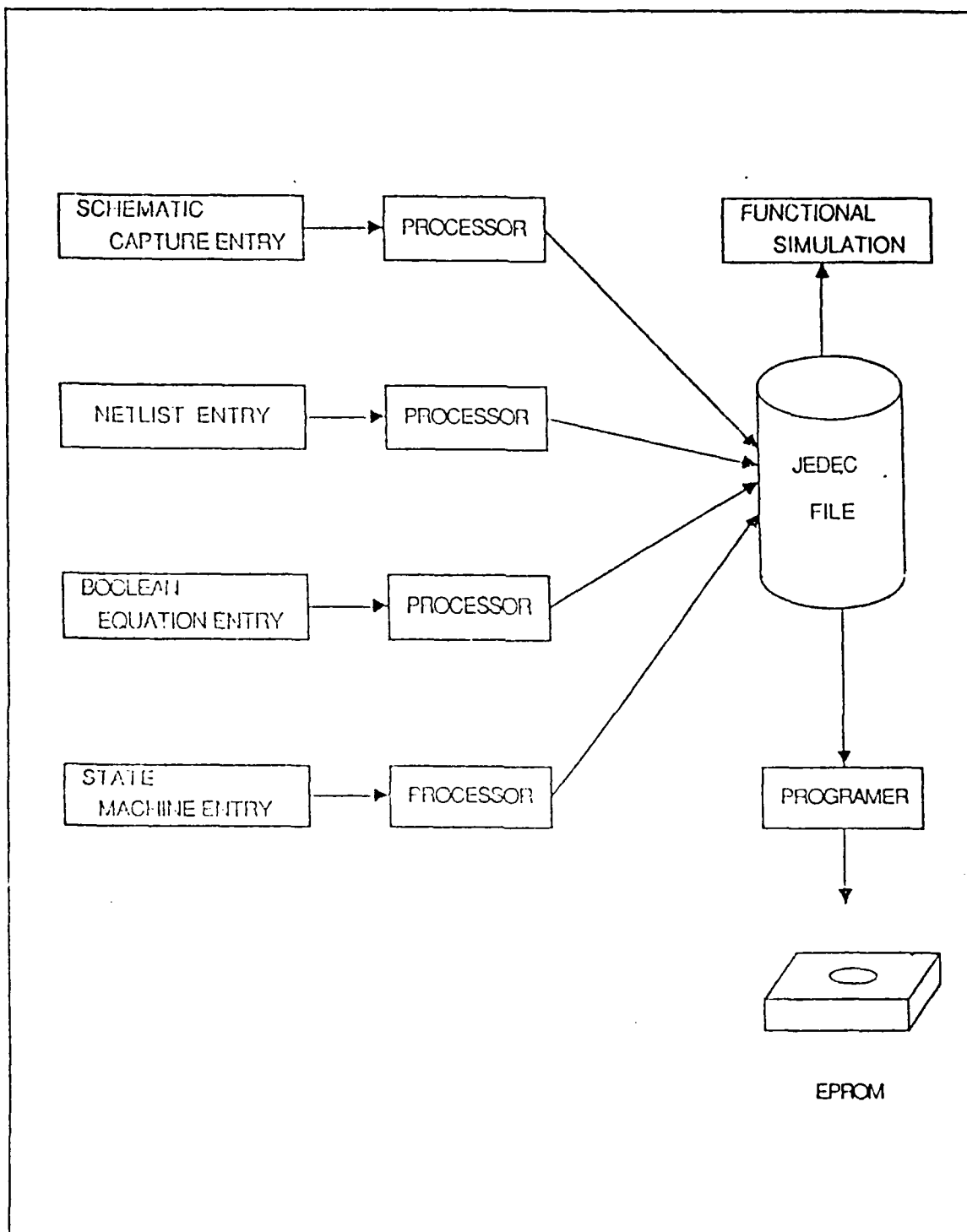


Figure 17. BLOCK diagram of EPLD design environment

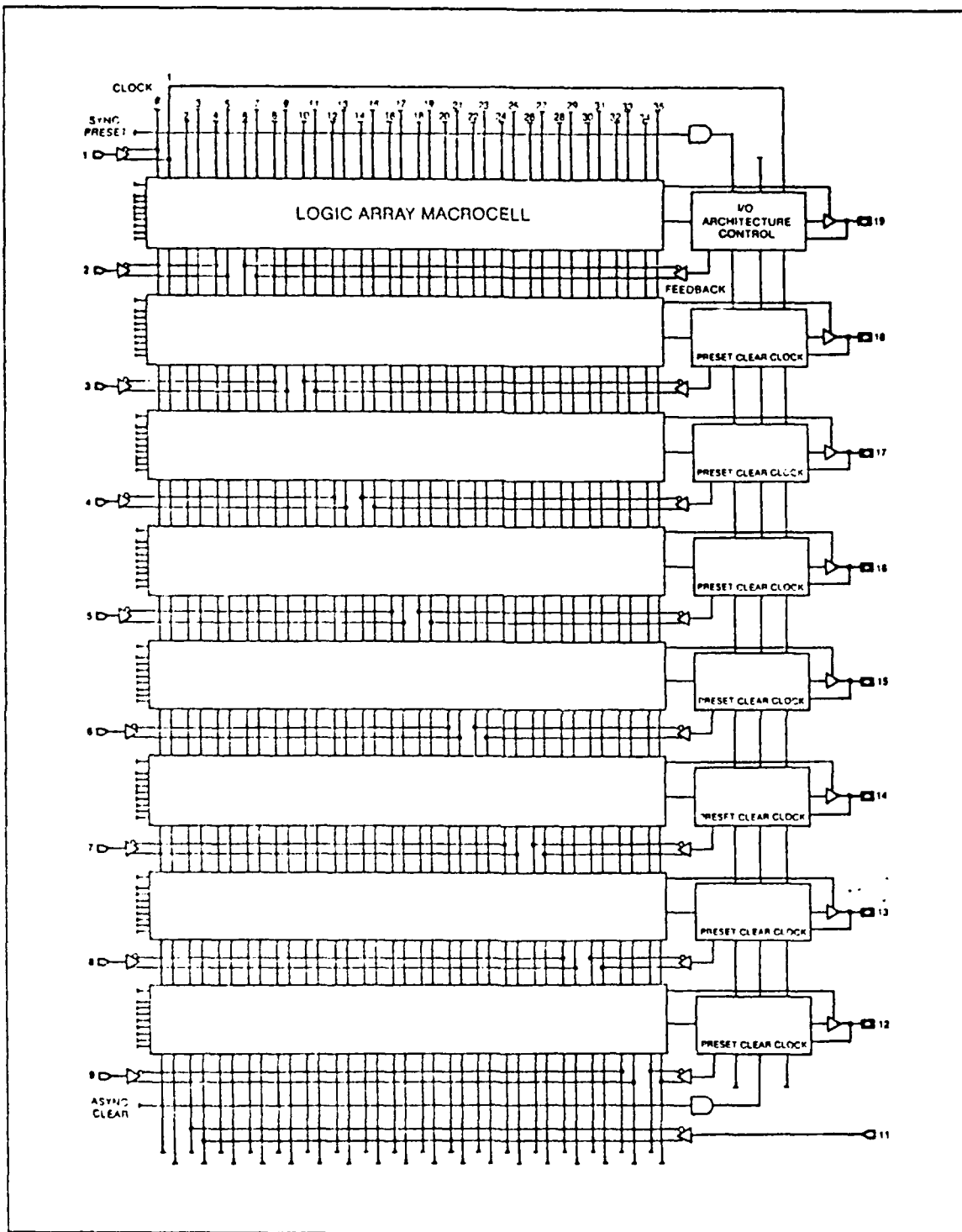


Figure 18. EP310 block diagram: [From Ref. 6]

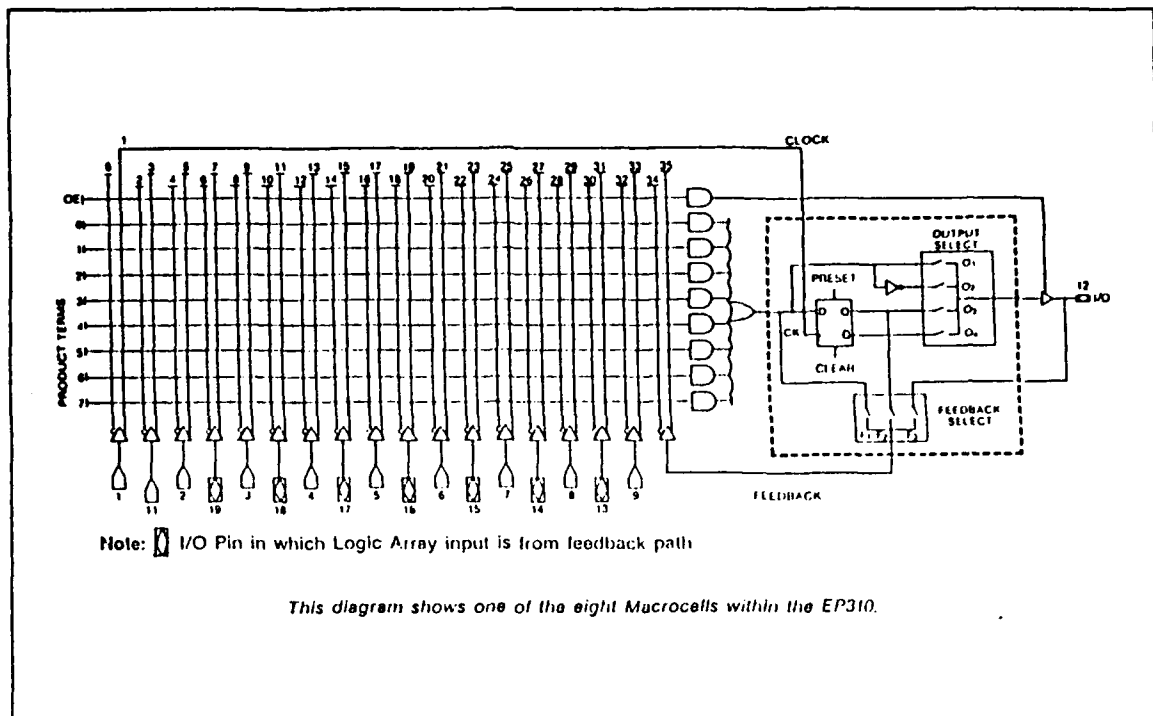


Figure 19. EP310 macrocell [From Ref. 6]

At each point of intersection in the product term, as shown in Figure 19, there exist an EPROM type programmable connection. Initially, all connections are made. This means that both the true and complement of the inputs are connected to each product term. Connections are opened, according to JEDEC file during the hardware programming process. Therefore, any product term can be connected to the true or complement input signals. When both the true and the complement connections of any input are left intact, a logical false results on the output of the product term. If both the true and complement connections of any input are programmed open, then a logical "don't care" results for that input. If all inputs of a product term are programmed open, then a logical true results at the output of the product term.

As shown in Figure 19, the outputs of 8 product terms are ORed together, and the output of the OR gate is fed as an input to the I/O Architecture Control Block. In the

I/O Architecture Control Block the signal from the OR gate is configured for register or combinatorial operation via Output Selection Switches. Both types of operations can produce inverted output. The feedback mode of the I/O Architecture Control Block can be programmed as combinatorial feedback, registered feedback, I/O (i.e, directly from the pin), or none, via the feedback selection switches.

Besides the normal macrocell product terms, there are additional Synchronous Preset and Asynchronous Clear product terms. These product terms are connected to all D-type flip-flops. When the Synchronous Preset product term is asserted HIGH, the output of the register will be loaded with a HIGH on the next LOW to HIGH clock transition. When the Asynchronous Clear product term is asserted HIGH, the output of the register will immediately be loaded with a LOW independent of the clock. An asynchronous clear assertion overrides a synchronous preset assertion.

B. DEFINE THE PROBLEM

Since an EPLD device is programmable, its configured structure varies with different implementations. It is desirable to model an EPLD device in VHDL so that it is independent to the design environment, i.e, does not depend on the design tools. The model can read in a user created JEDEC file, and configure its internal connection to perform the user specified function.

Besides the correct functional simulation, a model must also provide the correct timing simulation. When the timing of the model is violated due to register timing requirement, or bus collision, the simulator should be able to warn the user and report where the error occurred.

C. DECOMPOSITION OF THE EP310

In this research, a hierarchical structure approach was used to model the EP310. The top of the hierarchy is the EP310 itself. Based on Figure 18, the components in the next level down in the hierarchy are the Logic Array Macrocell, I/O Architecture Con-

trol Block, and the tri-state buffer. Below the Logic Array Macrocell is the Product Term. Except for the tri-state buffer, all the other next level elements are declared as components in the EP310's architectures as shown in Appendix A. The Logic Array Macrocell can be decomposed into 8 Product Term components. The I/O Architecture Control Block can be decomposed into the D-type flip-flop, the Output Select Unit, and the Feedback Select Unit as shown in Figure 19. Here, only a D flip-flop is constructed as a component below the I/O Architecture Control Block. The Feedback Select Unit and the Output Select Unit are implemented directly inside the I/O Architecture Control Block.

Besides these 4 components, the EP310 model also need a place to keep the timing parameters and a function "READ_310" to read in the JEDEC file. All these EP310 dependent functions and parameters were put into a package called EP310_PACK as shown in Figure 20.

Functions to convert the types are needed whenever different type signals are passed between different entities. A resolution function to resolve the multiple-source signal is also required. Because of the general usability of these functions for all parts of the EPLD, these functions are put into a package called EPLD_PACK, which is visible to all the necessary components as shown in Figure 20.

D. ESTABLISH DATA FLOW

After the hierarchy of an EP310 was established, it is necessary to reveal the signal flows between different components at different hierarchical levels. Externally, the EP310 chip can only see 18 data pins. Ten of them are input pins and the others are input output pins. These 18 data pins are the data path between the outside circuit and the EP310 internal components. Beside these signals, an EP310 model must also receive a JEDEC file from the outside to simulate the designed behavior. This is done by passing JEDEC file data through a generic port to the simulated entity.

As discussed previously in this chapter, the Logic Array Macrocell has 36 signal lines which are derived from 18 data pins. In the VHDL model, instead of using 36 signal lines, only 18 data lines were fed into the Logic Array Macrocell. These 18 data lines corresponding to the 18 data pins on the EP310. Doing it this way helps to reduce the excessive internal signals in the VHDL. The EP310 depending on the I/O architecture configuration can output signals through the input/output pins.

The Logic Array Macrocell accepts 18 input signals and produce two internal signals. One is the tri-state buffer output enable, and the other is the ORed signal fed into the I/O Architecture Control Block. The Logic Array Macrocell must also receive the corresponding JEDEC file information, i.e., 8 product term for each row of JEDEC file data.

There are three input signals to the I/O Architecture Control Block, one from the Logic Array Macrocell, one from the dedicated input pin (serve as synchronous clock), and the last one from the I/O pin. The I/O Architecture Control Block outputs two signals, one is the feedback signal, and the other is fed into the tri-state buffer. The I/O Architecture Control Block must also receive the corresponding JEDEC file information that used in configuring the switches in the two select units.

For a D flip-flop there are four input signals. One is the clock which is from the dedicated synchronous clock input pin. The second one is the synchronous preset which is from the synchronous preset product term. The last one is the asynchronous clear which is from the asynchronous clear product term. Although, there are no programmable connections in the D flip-flop, the JEDEC file informations is still needed to disable the assertion mechanics inside the D flip-flop, which will be discussed in the REGISTER TIMING section.

The Product Term was used inside the Logic Array Macrocell and the EP310. The Product Term has 18 inputs signals passed from the next higher hierarchy level and

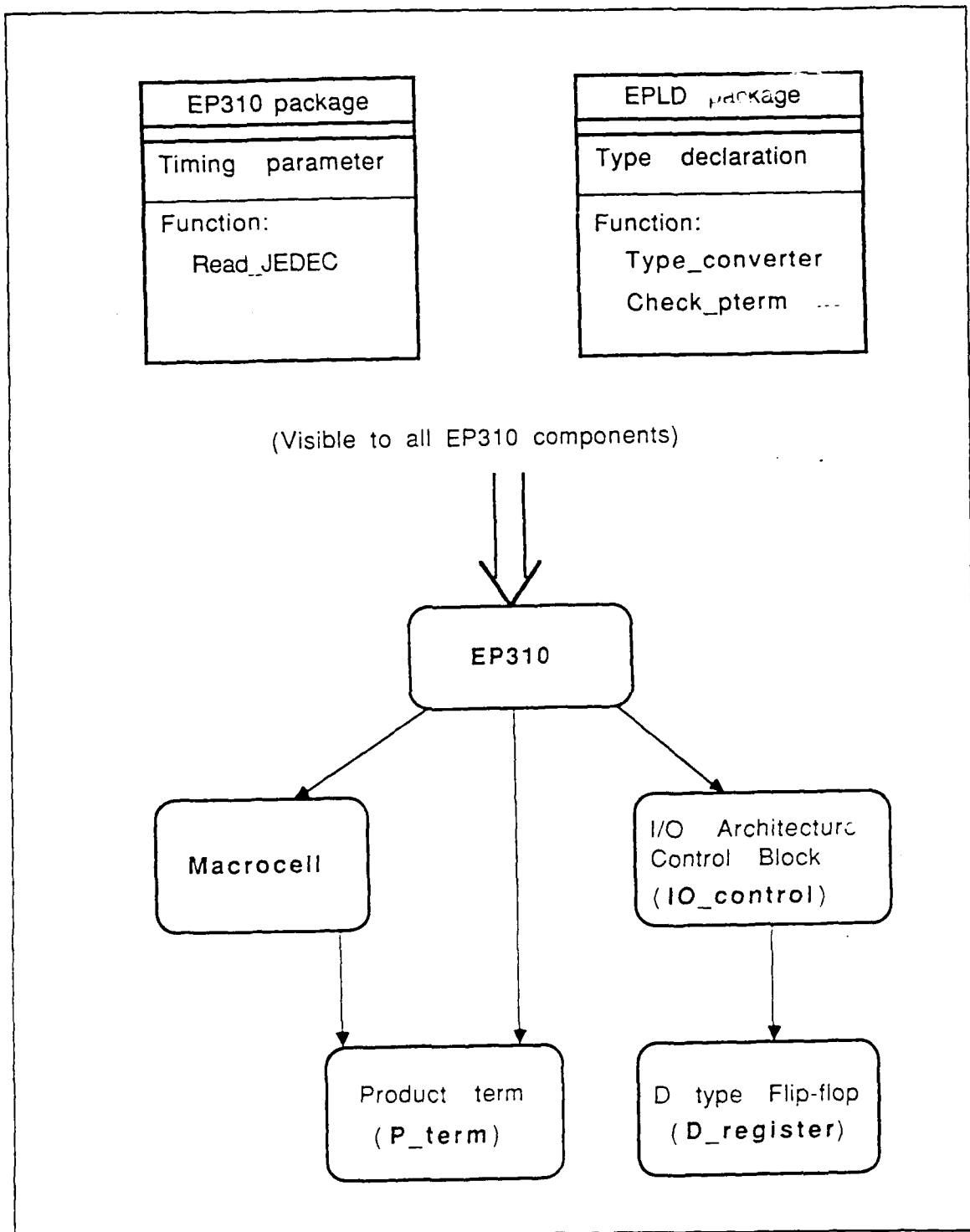


Figure 20. Decomposed EP310 hierarchical block diagram

produce one output signal. The Product Term must also receive the JEDEC file information from the next higher level entity in order to perform the user specified functions.

Figure 20 shows the decomposed hierarchical block diagram. As discussed above, the EP310 consists of 4 components, Logic Array Macrocell abbreviated as Macrocell, I O Control Architecture Control Block abbreviated as IO_control, Product Term abbreviated as P_term, and D flip-flop abbreviated as D_register.

The binding of component to entities is performed by configuration specifications. The configuration performs a component selection kind of function. With this feature, interchange of components from different technologies are possible. As discussed in Chapter II, VHDL configuration specifications appear in the declarative part of the block where the corresponding component are called. In certain cases, however, it may be more suitable to leave the configuration unspecified and defer such decision until the Kernel is built [Ref. 4: pp.1-9].

E. SIGNAL ASSIGNMENT

There are two ways to assign signals. One is to gather signals with common properties in an array. The other is to treat the signals separately. The benefit of using a signal array is its simplicity. But, there is a drawback. That is, each time when one of the elements in the array is activated, the whole array is activated. The processes associated with the elements of the array will be activated too often in a complex design. This costs a lot of simulation time. On the contrary, the single signal assignment may be more efficient in reducing the simulation time. But, it will make the model complex and hard to comprehend. In this study, the EP310 data pin signals are defined separately in order to reflect the real chip pinout; the rest of the signals are implemented by using the signal array.

F. JEDEC FILE INFORMATION TRANSFER

As mentioned before, the EPLD model needs to read in the JEDEC file information in order to perform the designed function. There are three methods to transfer the JEDEC file information from outside to an entity model:

1. Assign information via **generic**.
2. Passing information via **port**.
3. Passing information by predefining it as a **constant** in the package, and make it visible to the entities.

Since the JEDEC file depends on the user's implementation, it can vary. It is not a good approach to declare the JEDEC file data as constants inside the package. If signals were used to transmit the JEDEC file data, the total number of signals including the implicit signals will be too large. This will affect the total simulation time, since every signal driver of this large set has to be checked at each delta cycle.

Due to the reasons discussed above and the nature of the JEDEC file, passing information via the **generic** is preferred. In this research, using generic to transfer JEDEC file information were adopted. The JEDEC file name is fed in via the EP310 generic at one level higher than the EP310 entity. In the EP310 declaration part a function called `READ_JEDEC` is used to read in the necessary JEDEC information. An example of using generics to pass the JEDEC file data to EP310 entity is illustrated on the next page.

```

-- inside package EP310_PACK declaration.

function READ_JEDE(file_name: in string)

    return jedec_file_data;

-- inside the TEST_BENCH body.

architecture demo of test_bench is

    component EP310 generic(JEDEC_file_name);

        port(pinout_specification);

        -- other statement.
        -----
        -----

begin

    EP310 generic map(user_defined_JEDEC_file)

        port map(pinout_specification);

        -- other statement.
        -----
        -----

end

```

```

-- inside the EP310 entity body

architecture STRUCTURAL of EP310 is

```

```

    constant BIT_MAP : jedec_file_data

        := READ_JEDEC("JEDEC_file_name");

```

The constant "BIT_MAP" is declared to have a "jedec_file_data" type, and its value is assigned by the returned value from the function READ_JEDEC. After the BIT_MAP

is initialized, its subcomponents are passed via generics to the corresponding components below the EP310. Note that the "jedec_file_data" type is not defined here. Refer to Appendix A for the actual declaration and the exact program.

G. MULTIPLE LEVEL LOGIC

Because not all pins on the chip will be used, it is necessary to introduce a signal with three-level logic to represent the unconnected situation. The EP310 model uses a new type TRI, which as shown below has a three level logic: '0', '1' and 'U'.

```
type TRI is ('U','0','1');
```

The 'U', depending on the location of the signal, means unconnect or undefined.

Since the VHDL is a strongly typed language, when constructing a multiple state design, it is necessary to include a type conversion function to pass different type signals between different processes. The following is a conversion function example used in the EP310 model.

```
function tri_to_bit(inbit: tri) return bit is
begin
    if inbit='1' then
        return '1';
    else
        return '0';
    end if
end tri_to_bit;
```

Note that, the conversion function will return a value '0', if the input is 'U'.

II. BUSSED SIGNAL

Shown in Figure 21 is an example where the signal "local" is driven by two sources. The sources of the signal "local" can come from the output of the tri-state buffer or from the "io_pin". Below is the statement that can model this connection.

```
local<= output after tod when enable='1' else
    io_pin after tio+tin;                --tio+tin is input delay
```

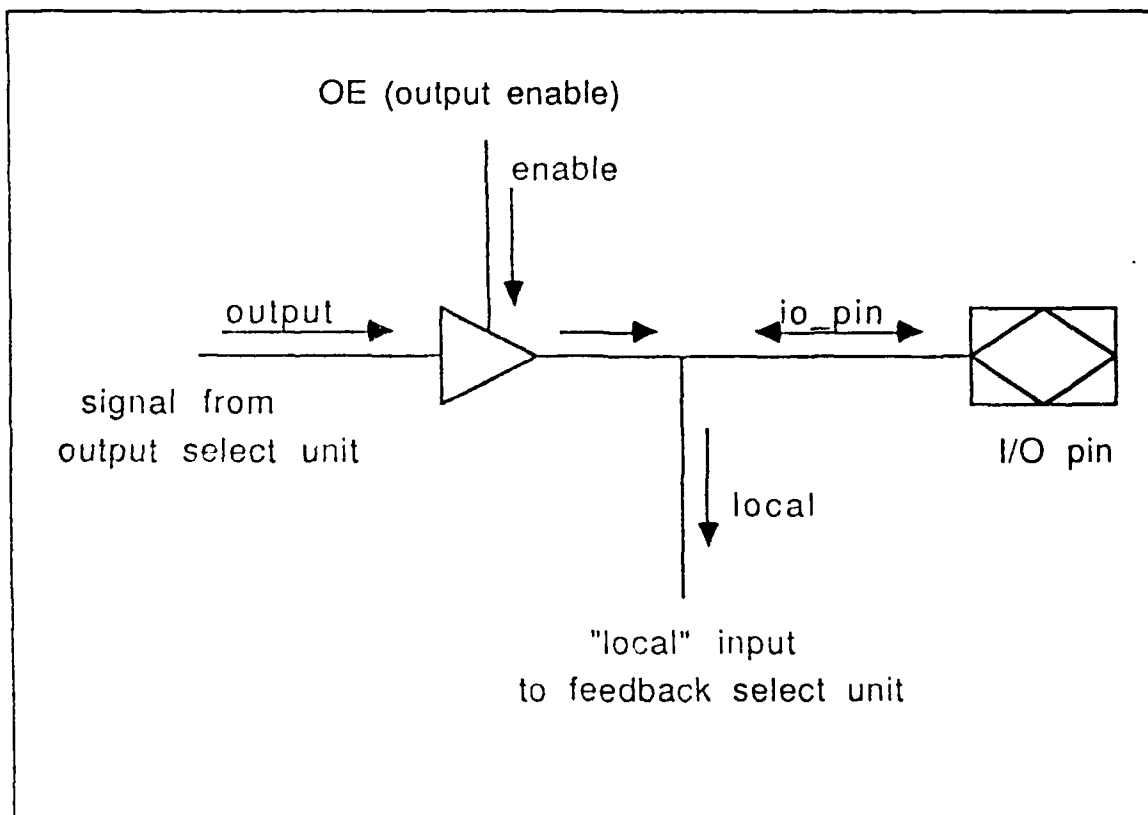


Figure 21. I/O primitive signal flow diagram

Note that the statement above has implied that the "output" signal has higher priority than the "io_pin" signal. This is reasonable, since the "output" signal drives not only the "local" signal but also the "io_pin" signal. Only when the tri-state buffer is disabled, will the "io_pin" signal value be assigned to the signal "local".

As mentioned before, the "io_pin" is a bi-directional pin driven by two sources, one is the tri-state buffer output, and the other is an external user input. It is driven in a time multiplexing sense. In order to model this circuit, it is necessary to declare the "io_pin" signal as a resolved signal. A resolved signal is a signal that has an associated resolution function. Resolution function is a user defined function that computes the value of a resolved signal from its multiple sources.

There are two ways to declare a resolved signal. One is by adding a resolution function to the front of the type, such as:

```
signal io_pin: inout RESOLVER tri;
```

The other way is to declare the signal with a resolved subtype, such as:

```
subtype tri_state is RESOLVER tri;  
signal io_pin: tri_state;
```

The resolution function used here is called the RESOLVER. Basically, the RESOLVER will gather all the sources of the declared signal, compute it according to user defined rules, then return the resolved value. Below is the implementation of the function RESOLVER.

```
function RESOLVER(signal inputs: tri_vector)  
return tri_state is  
    variable resolved_value: tri_state:='U';  
    variable flag: integer:='0';
```

```

begin
    for i in inputs'range loop
        if inputs(i)/='U' then
            flag:= flag + 1;
            resolved_value:= inputs(i);
        end if;
    end loop;
    assert flag <= 1
        report "io_pin bus collision."
        severity FAILURE;
    return resolved_value;
end RESOLVER;

```

In the signal assignment statement the io_pin is assigned as:

```

io_pin <= output after tod when oe = '1' else
    'U' after tod when oe= '0' else
    'U';

```

Note that if the corresponding tri-state buffer is enabled, i.e., oe='1' and, in the meantime, the user gives io_pin another assignment from the top entity, then the signal will have two active sources. It means that the source values are not all 'U'. In this case, the resolution function shown above will assert the data collision message. On the other hand, if only one source is active, the resolution function will output the active source value and assign it to the io_pin without the violation message.

1. PRODUCT TERM INTERNAL CONNECTION

As mentioned previously the EPLD logic array has programmable internal connections. The connections were made according to the corresponding JEDEC file spec-

ification. In this research a function, called `check_Pterm`, is used to generate the correct functional output. The `check_Pterm` function has two type of inputs:

1. Eighteen signals from the feedback lines shown in Figure 19.
2. Thirty-six internal connection specification defined as `P_string` from the JEDEC file.

The `check_Pterm` function generates the correct value according to the specification and the conventional logic array rules discussed in the previous section. The main mechanism of this function is to decide whether the input signal should be included by examining the true and complement connections of the signal in the JEDEC file. Shown below is a portion of the decision statements used in the `check_Pterm`. The statements are written in VHDL like pseudo code. Refer to the package `EPLD_PACK` of Appendix A for the full exact detail:

```
-- begining of the decision statement.
output:='1';
i:=1;
while i<=P_string'length loop
    if(P_string(2*i-1)='0' and P_string(2*i)='1') and
      (input_signal(i)='U' or input_signal(i)='1') then
        output:='0'
        exit;
    end if
    -- other decision statements.
end loop;
return output;
```

The "`P_string`" in the above example is an array which contains the programmable AND array internal connection data. The statement "`P_string (2*i-1)='0' and`

$P_string(2*i) = '1'$ will check the connections to see whether only the complemented connecting point shown as $P_string(2*i-1)$ exists. If the complemented connecting point exists and if the corresponding "input_signal(i)" is '1' or 'U' (unconnected), the output will be '0'. With statements similar to the above example, the other three situations are checked to produce the correct output. These three situations are: a) both the true and the complement connecting points exist, b) both points do not exist, or c) only the true connecting point exists. See the function `check_Pterm` of Appendix A for the details.

J. EPLD TIMING SIMULATION

As discussed in a "EPLD timing simulation" paper [Ref. 7], the timing diagram of a general EP310 is shown in Figure 22. The delay time through logic array (**tlad**) was modeled as a constant. The rest of the timing parameters were like those found in the conventional logic circuit. Below are the description of the timing parameters shown in Figure 22.

1. **tin**- input pad and buffer delay.
2. **tio**- I O input pad delay. This delay need to be added to **tin** when an I O pin is used as input.
3. **tod**- output buffer and pad delay.
4. **txz**- time to tri-state output delay.
5. **tzx**- tri-state to active output delay.
6. **tlad**- logic array delay.
7. **tlade**- enhanced logic array delay.
8. **tsu**- register setup time.
9. **th**- register hold time.
10. **tcclr**- asynchronous register clear time.
11. **tcldre**- enhanced asynchronous register clear time.
12. **tics**- system clock delay.
13. **tic**- clock delay.
14. **tice**- enhanced clock delay.
15. **tfd**- feedback delay.

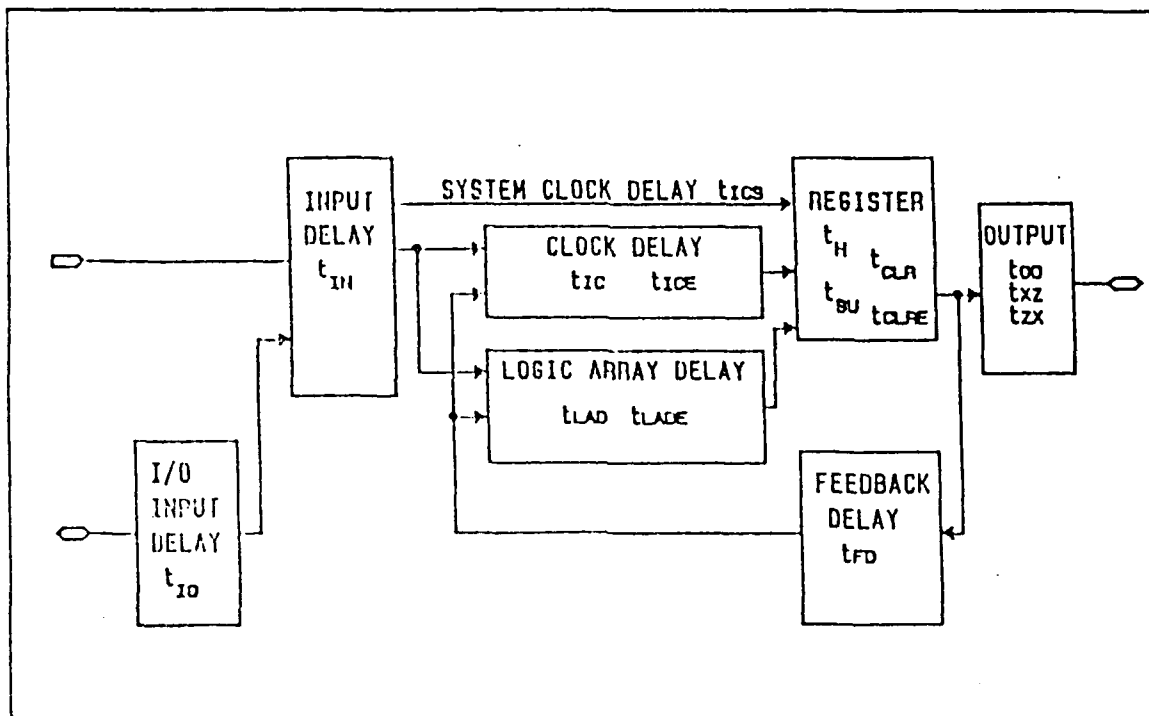


Figure 22. EPLD timing block diagram [From Ref. 7]

16. t_{ch} - minimum clock duration, when clock is high.

17. t_{cl} - minimum clock duration, when clock is low.

Since a EP310 is decomposed into four components shown in Figure 20, it is necessary to assign each component with its associated timing parameter. Figure 23 shows the decomposed components with their timing parameters. All the timing parameters used in this research can be found in [Ref. 7].

1. REGISTER TIMING

The most important timing problem related to a register is its *setup time*, *hold time*, and *minimum pulse width*. For example, consider the clocked register in Figure 24. It is a common requirement that the data input be stable for a duration of time prior to the clock transition that strobes the data into the register. This requirement is known as the setup time of the data relative to the clock. A similar requirement

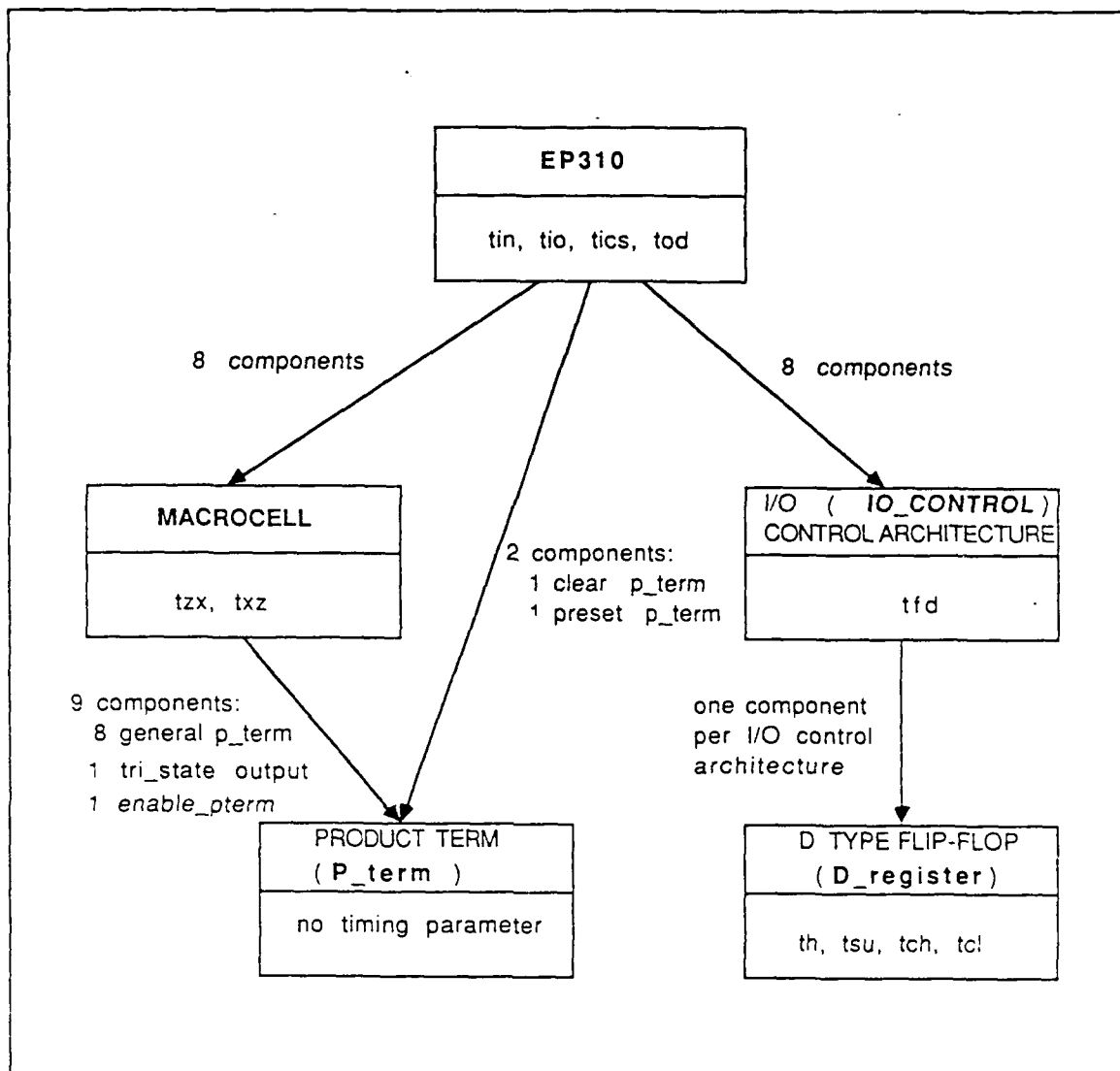


Figure 23. EP310 hierarchical block diagram with time parameter

states that the data should remain stable a minimum amount of time after the clock transition which is known as the hold time requirement.

Figure 24 shows the input specifications for the register in a typical EPLD. The specification says that (1) DATA should be stable for *tsu* nanoseconds before CLOCK rises, (2) DATA should be stable for *th* nanoseconds after CLOCK rises, and (3)

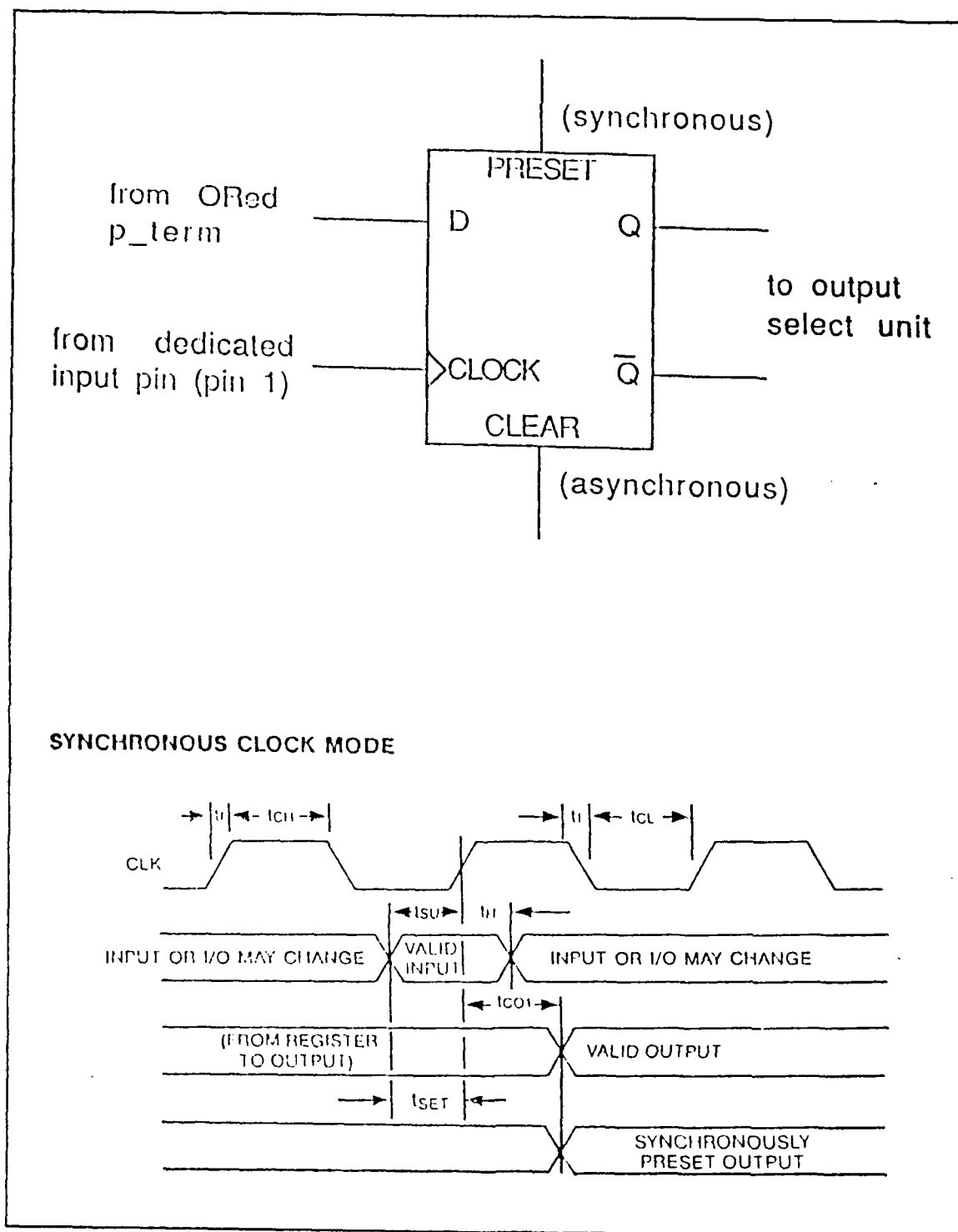


Figure 24. D register timing diagram

CLOCK should have a minimum duration at the high level of at least `tch` nanoseconds. The following assertion statement in VHDL checks the setup time specification of the D register of the EP310:

```
assert not( not clk'stable and clk='1' and (not d'stable(tsu)) and
           clear='0' and (O3='0' or O4='0' or F2='0'))
report "Setup Time Failure";
```

Using the DeMorgan's theorem, one could convert the assertion statement to a simpler form

```
assert ck'stable or ck='0' or d'stable(tsu) or clear='1' or
       (O3='1' and O4='1' and F2='1')
```

In the above example the attribute `D'STABLE(tsu)` will be TRUE if and only if signal D has been stable for `tsu` time units already. Since this D register has an asynchronous clear, it is necessary to include "clear='0'" in the assertion statement. Otherwise, when the "clear" is set to one, the D register will resume the synchronous clock cycle and produce the errorous result. The reason for checking the switches O3, O4, and F2 in Figure 19 is to see if the register is selected to operate. If the macrocell is dedicated to perform combinational logic, then the assertion will be disabled.

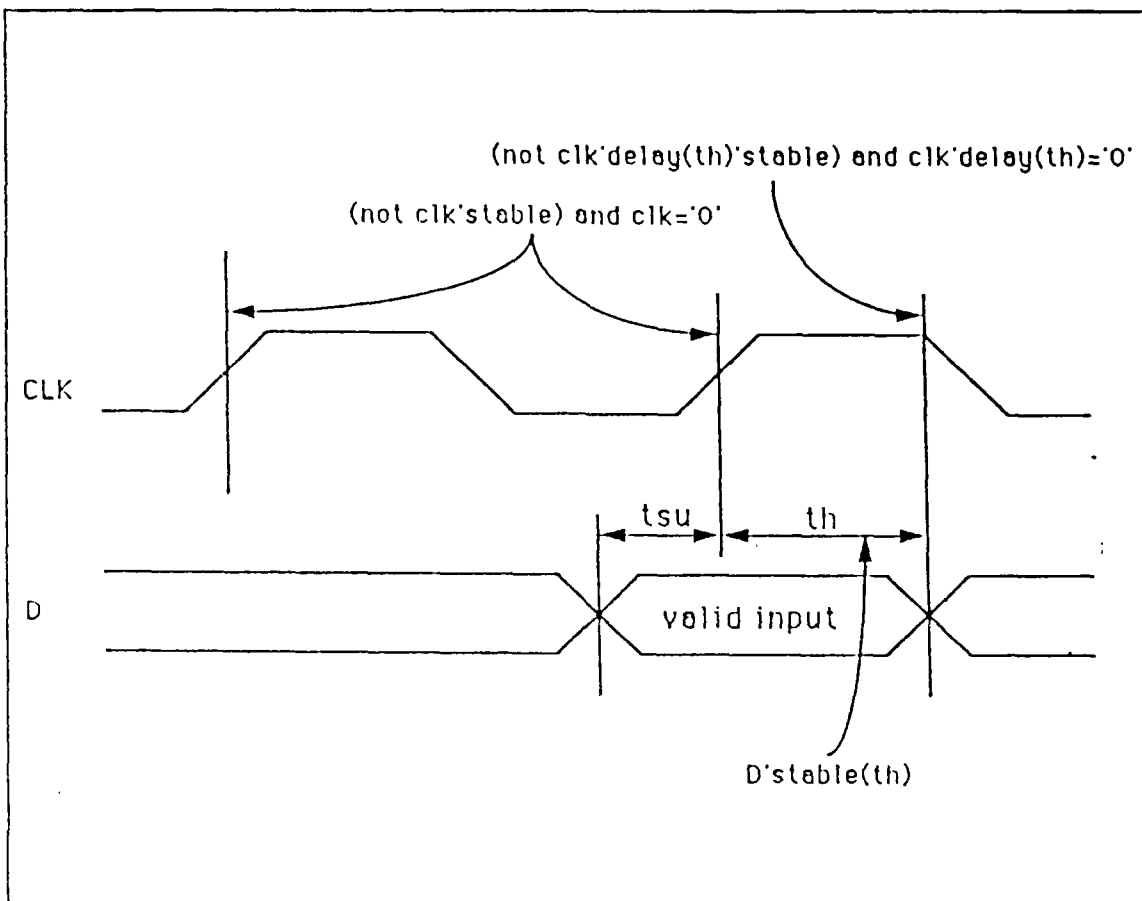


Figure 25. Hold time timing diagram

The hold time can be checked by using the following VHDL statement:

```
assert ck'delayed(th)'stable or ck'delayed(th)='0' or d'stable(th) or
      clear='1' or (O3='1' and O4='1' and F2='1')
report "Hold Time Failure";
```

The statement will check the stability of data "d" after the necessary hold time (th). The hold time start from the rising edge of the clock and end at the falling of the clock as shown in Figure 25. If the evaluation of the statement are false then the error will be reported, which means the flip-flop hold time is not satisfied.

Similarly, the minimum pulse-width can be checked by using the following statement:

```
assert ck'stable or ck='1' or ck'delayed'stable(tch) or (03='1' and 04='1'
and F2='1')
report "Minimum pulse width failure";
```

K. COUNTER

An EP310 was configured to implement a 7-bit counter. The JEDEC file was generated by using the Altera logic design tool. The model was tested as a 7-bit counter. The simulation starts from a top entity called the "TEST_BENCH". The code of the "TEST_BENCH" and its architecture body are shown in Appendix B. The simulation result are also shown in Appendix B. Basically, the "TEST_BENCH" calls the EP310 entity, provides the signals to the EP310's input pins, and the the external JEDEC file name to the EP310's generic. The user can control the simulation time via the generic of the "TEST_BENCH".

IV. MODELING THE EP1800

A. INTRODUCTION OF THE EP1800

The EP1800 like the EP310 discussed in Chapter III is also an Erasable Programmable Logic Device with a larger number of macrocells. In the EP310 there are about 300 gates, but in the EP1800 there are about 2100 gates [Ref. 6: pp. 2-4]. The EP1800 has a classic programmable AND array just like the EP310. But, unlike the EP310 which has only one D-type flip-flop, the register inside the EP1800 can be programmed into a D-type, T-type, JK type, or RS type flip-flop. Each register can be clocked asynchronously on an individual basis or synchronously on a banked register basis [Ref. 6: pp. 2-6].

The block diagram of an EP1800 is shown in Figure 26. Externally, the EP1800 provides sixteen dedicated data inputs, four of which may be used as the system clock inputs. There are 48 I/O pins which can be individually configured to be input, output, or bi-directional pins.

Internally, the EP1800 contains 48 macrocells. As shown in Figure 27 and Figure 28, each macrocell contains three basic elements: a logic array, a selectable register element, and a tri-state I/O buffer. All the combinatorial logics are implemented within the logic array. For register applications each macrocell provides one of four possible flip-flop operations: D, T, JK, and SR.

The EP1800 is partitioned into four identical quadrants as shown in Figure 26. Each quadrant contains 12 macrocells. The macrocell input signals come from the EP1800 internal bus structure. The macrocell output may drive the EP1800 external pins as well as the internal buses. Sixteen of the EP1800's 48 macrocells offer increased speed performance through the Logic Array. These "Enhanced Macrocell" can be uti-

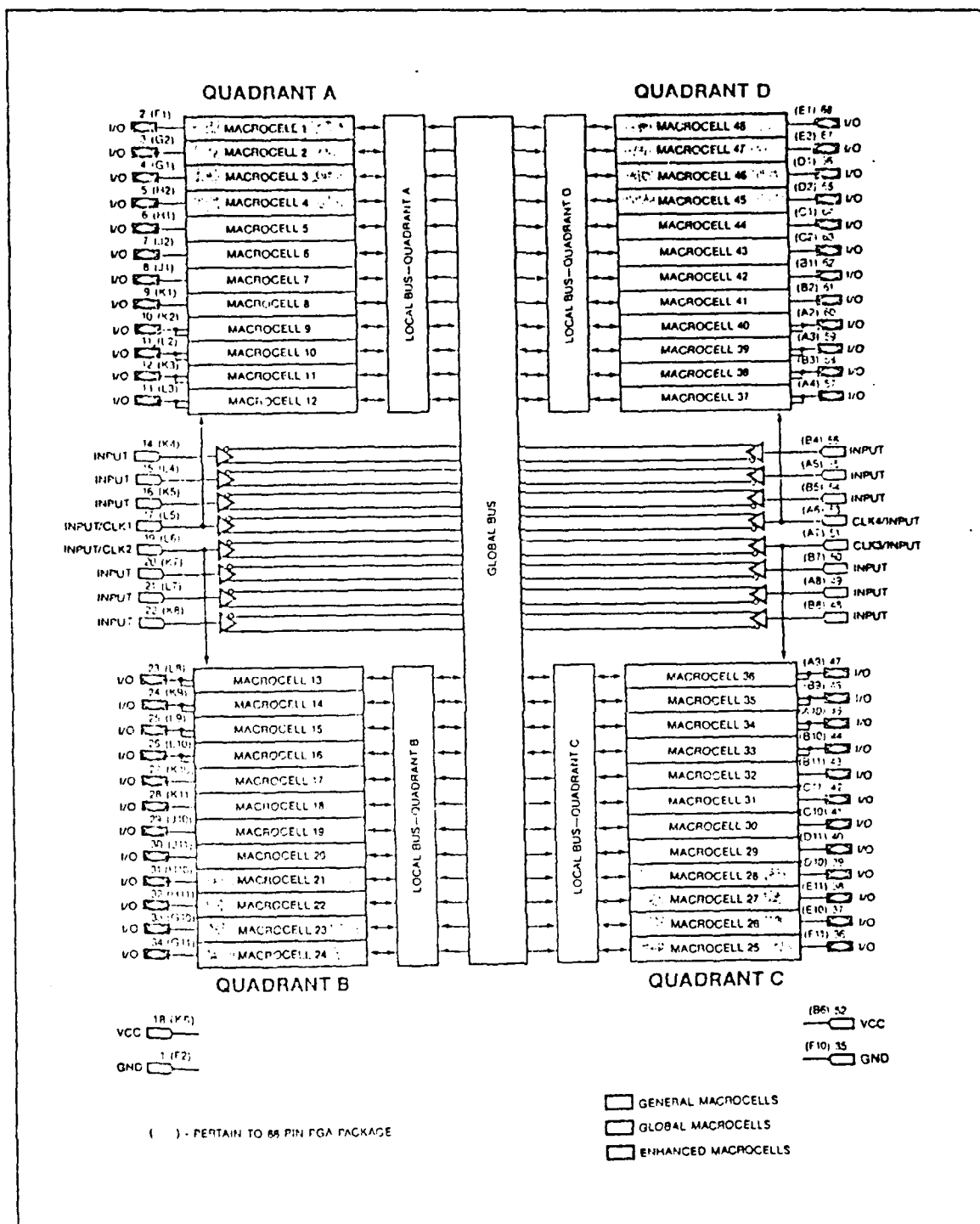


Figure 26. EP1800 block diagram [From Ref. 6]

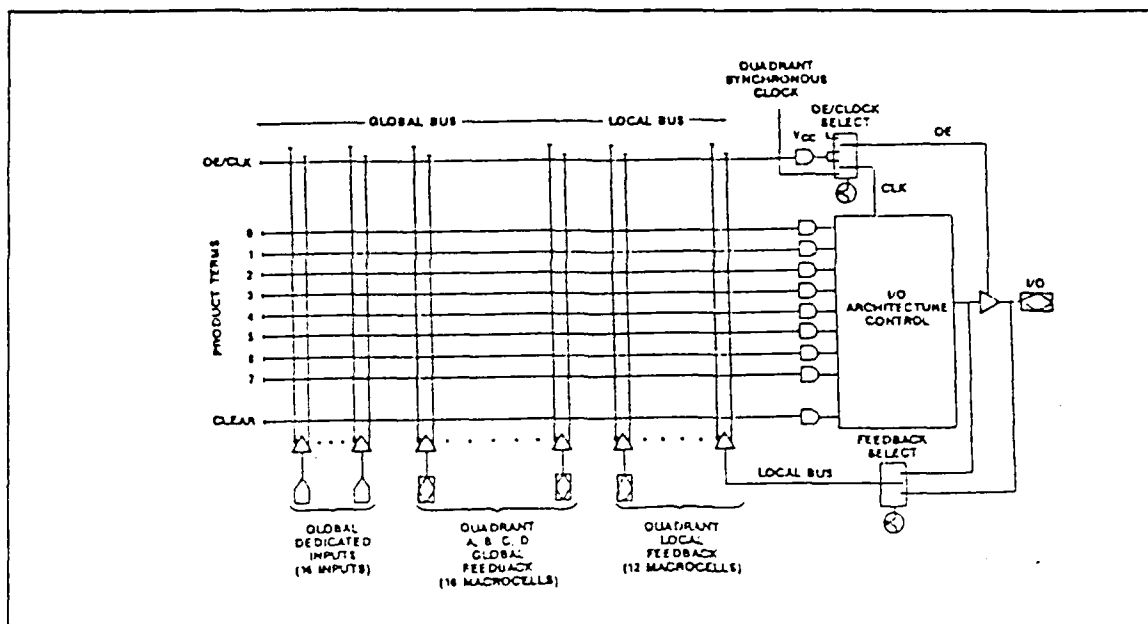


Figure 27. Local macrocell: [From Ref. 6]

lized in places where combinatorial logic path delay is critical. There are four Enhanced Macrocells for each EP1800 quadrant as shown in Figure 26. A detailed delay timing specification of these Enhanced Macrocells are listed in the file EP1800_PACK in Appendix A.

There are other sixteen "Global Macrocells" that can provide dual functions. These Global Macrocells shown in Figure 28 implement *buried logic functions*, and at the same time serve as dedicated input pins. Thus, the EP1800 may have an additional 16 input pins yielding a total of 32 inputs. The Global Macrocells have the same timing characteristics as the General Macrocells [Ref. 6: pp. 2-7].

Each of the EP1800 internal flip-flops can be clocked independently or in user-defined groups. Any input or internal logic function may be used as a clock. These clock signals are activated by driving the flip-flop clock input with a clock buffer primi-

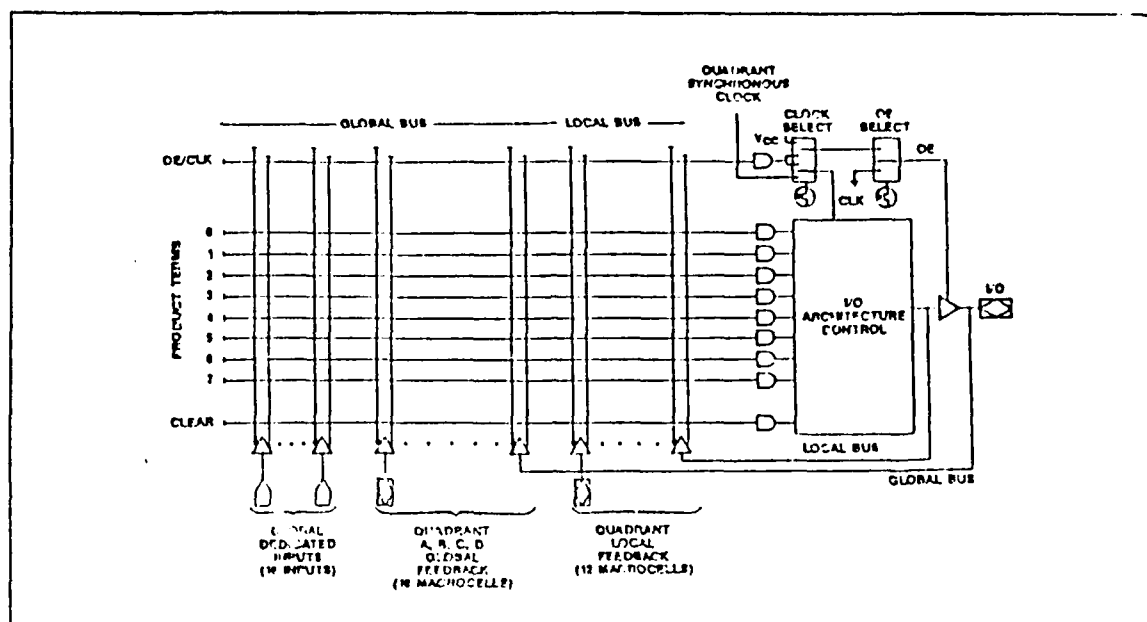


Figure 28. Global macrocell: [From Ref. 6]

tive (CLKB). In this way, the flip-flop can be configured for either positive or negative edge triggered operations.

Four dedicated system clocks, CLK1 through CLK4, provide clock signals to all the flip-flops. System clocks are fed directly from the EP1800 external pins. With this direct connection, a system clock imposes the shortest delay than internally generated clock signals. There is one system clock per EP1800 quadrant. When using system clocks, the flip-flops are positive edge triggered, i.e., data transitions occur on the rising edge of the clock.

B. DECOMPOSITION OF THE EP1800

Just like the EP310, the hierarchical structure approach is used to decompose the EP1800. The top level is the EP1800 itself. Based on Figure 26 the next level down in the hierarchy are the Quadrants. Below this level is the Global Macrocell and the Local

Macrocell. The hierarchical levels lower than macrocells are similar to those structures in the EP310.

The EP1800 model also needs a function `READ_JEDEC` to accept the JEDEC file and a few other necessary functions to implement the real physical device. All these functions together with the timing parameters and the defined types are stored in a package called `EP1800_PACK` as shown in Figure 29.

The `EP1800_PACK` and the EPLD packages discussed in Chapter III are visible to all the necessary components. The total EP1800 hierarchical structure is illustrated in Figure 29.

C. ESTABLISH DATA FLOW OF THE MODEL

After the hierarchy of the EP1800 is established, it is necessary to identify the signal flows between different components.

Looking at an EP1800 chip from outside, only 60 data pins and 4 dedicated system clock pins are visible as shown in Figure 30. Therefore, at the top level of the entity EP1800, there are 64 signals in its port declaration. These 64 signals serve as the signal paths between outside circuit and the EP1800 internal components. Besides these signals, an EP1800 model must also receive a JEDEC file from the outside to simulate the user-designed behavior. This is done by passing the JEDEC file name through a generic port to the simulated entity just like the method used for EP310.

Inside the EP1800 entity, there are 4 quadrants. These 4 quadrants are symmetric in some sense, i.e., two of them are upside down. Therefore, there is only need to build one quadrant model. The remaining quadrants can be implemented by using the model directly or modifying the generic with the reverse function. The details are described in the EP1800 architecture body of Appendix A.

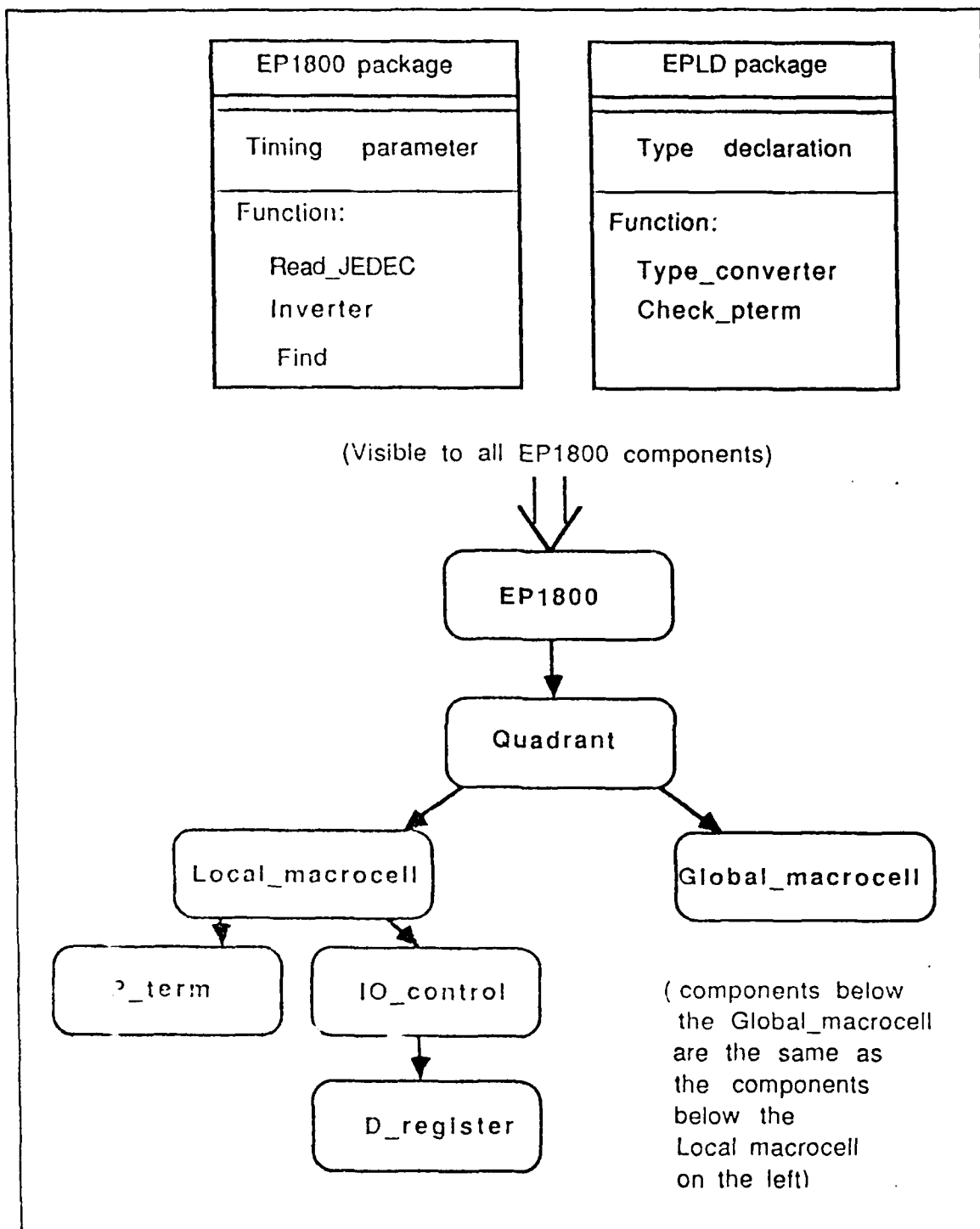


Figure 29. EP1800 hierarchical model diagram

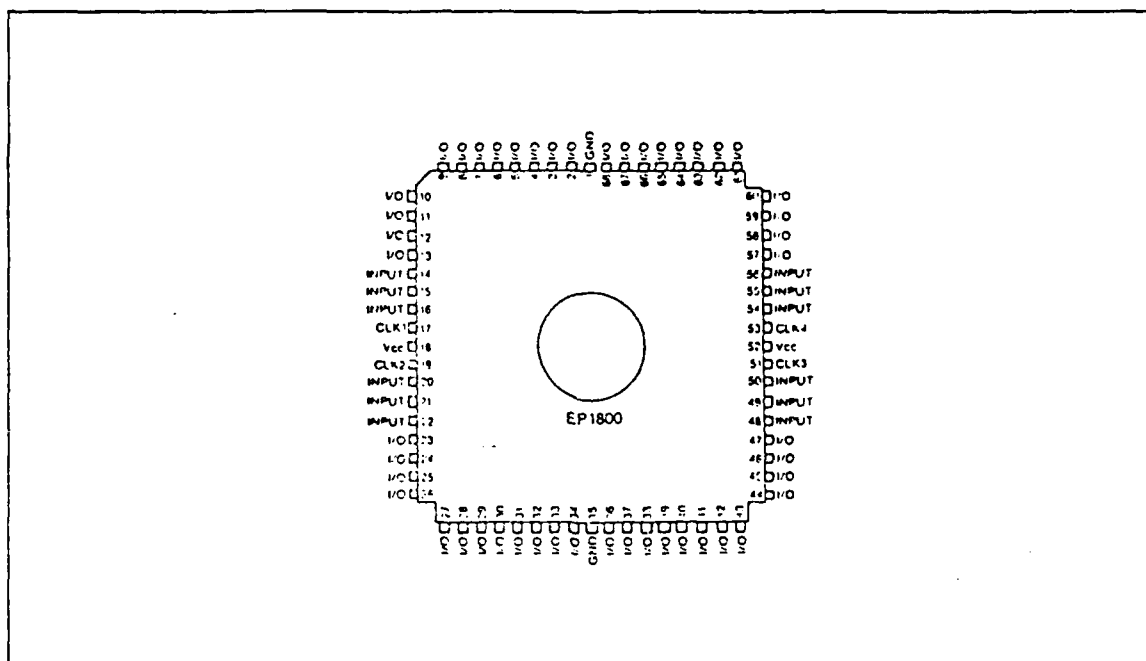


Figure 30. EP1800 chip overview: [From Ref. 6]

The signals going into each Quadrant include the global bus inputs and a system clock input. Four of the Quadrant output signals are from the Global Macrocell tri-state buffer outputs. The rest of the 8 signals are bi-directional General Macrocell I/O pins.

There are 12 macrocells inside each Quadrant. These macrocells are classified as Global Macrocell, General Macrocell, and Enhanced Macrocell. The structure of the General Macrocell is the same as the Enhanced Macrocell. The only difference between these two macrocells is that the Enhanced Macrocell has a shorter time delay. Therefore, when modeling the EP1800, it is only necessary to build the General Macrocell modules and the Global Macrocell module. The Enhanced Macrocell module can be yielded by changing the timing parameters of the General Macrocell.

The macrocell accepts bus inputs and a system clock input. The bus inputs consist of 44 input signals. The macrocell outputs consists of two signals. One comes from the

I/O Architecture, and the other is a tri-state output enable signal. The internal structures of the Global Macrocell and the General Macrocell are identical except that the General Macrocell has a feedback select switch and the Global Macrocell has an output enable switch. The total numbers of signal-flow control switches are the same in both type of macrocells.

The I/O architecture in the macrocell accepts the ORed logic array signal, an input clock signal, and a register clear signal. The ORed logic array signal is generated from 8 ORed logic-and-arrays similar to those in the EP310 structure. The input clock signal, depending on the clock select switch, is either from the quadrant synchronous clock or from the enable clock logic-and-array output.

The internal structure of the I/O architecture consists of a D-type flip-flop, logic gates, and 3 signal flow control switches. Because of the non-disclosure agreement with the Altera corporation, the detail structure inside the I/O architecture is not discussed. In Appendix A, the I/O architecture module of the EP1800 only shows the entity declaration part.¹

All the EP1800 internal connecting point information in the JEDEC file are passed to the entity via the corresponding entity generic. The parameter in the generic of EP1800 is a character string which can be any user-created JEDEC file name. After the JEDEC file name is passed into the body of the EP1800, the function READ_JEDEC will read in the JEDEC file data stored in the JEDEC file name. The function will assign this composite type (EP1800_TYPE) JEDEC file data to a constant call BIT_MAP. Then, like that done for the EP310, the corresponding elements of this composite constant are passed to the corresponding lower level components. For more details please see Appendix A.

¹ For more information please contact Prof. CHIN-HWA LEE, Naval Postgraduate School, Monterey, CA 93943.

D. GENERAL MACROCELL AND ENHANCED MACROCELL

As mentioned previously, the General Macrocell and the Enhanced Macrocell have the same structure but different timing parameters. In order to make the model size more compact and to avoid the redundant VHDL program, these two types of macrocells use the same architecture body. The way to define whether the macrocell is a General Macrocell or an Enhanced Macrocell is by passing different timing parameters via generic. For Enhanced Macrocell this would look like:

```
ENHANCED: LOCAL_MACROCELL generic map(P_ARRAY(1 to 10),  
                                       IO_ARRAY(1), tlade, tclre, tic)
```

where the "tlade", "tclre", and "tic" are the enhanced timing parameters. The "P_ARRAY" and the "IO_ARRAY" are array types. They contain the corresponding product terms and I/O select unit JEDEC file data. The word "enhanced:" in front of the local_macrocell is referred to as the label of this component instantiation. If there are more than one identical component instantiation in the same block level, the labels with different names must be used in front of these instantiations. The General Macrocell and Enhanced Macrocell have the same expression but different generic values, i.e., the timing parameters.

E. THE REUSABLE QUADRANT MODEL

Although the Quadrants inside the EP1800 have the same structure, the pin assignment combined with the ordered JEDEC file data makes the generic map assignment of Quadrant-B and Quadrant-C in the reverse order. In order to reuse the same Quadrant model, in this work, two "reverse functions" are built. Both "reverse functions" can reverse the order of the input array. Below is one of the reverse function used in the EP1800 model.

```

function REVERSE(A: in IO_array_type) return IO
_array_type is
    variable rev_array: IO_array_type( 1 to A'length);
    variable c: POSITIVE := 1;
begin
    for i in A'REVERSE_RANGE loop
        rev_array(c):= A(i);
        c:= c+1;
    end loop;
    return rev_array;
end REVERSE;

```

Note that the attribute "A'REVERSE_RANGE" will output the A's range in the reverse order. That is, the attribute will have the range A'RIGHT **downto** A'LEFT if the range of A is ascending, or A'RIGHT to A'LEFT if the range of A is descending. These two reverse functions are put into the package EP1800_PACK.

When using the Quadrant B and C, the reverse function is placed in front of the generic parameter P_array, and IO_array as follows

```

QUADRANT_B:
    QUADRANT generic map(REVERSE(P_array(range)),
                        REVERSE(IO_array(range)))
    -- port map specification
    -----

```

Note that the variables inside the above two REVERSE functions have different types. In spite of the different types of this two arrays, the VHDL will automatically find the

corresponding matching function according to the input parameter type. In VHDL this kind of functions are called *overload* functions.

F. THE EP1800 BUS STRUCTURE

The internal bus structure of the EP1800 is more complicate than that of the EP310. Basically the EP1800 bus can be divided into two kinds, *local bus* and *global bus*. The local bus, as the name implies, only provides a signal path between macrocells within the same quadrant. The Global bus on the other hand provide the signal path to all the macrocells inside the EP1800 chip.

The local bus consists of 12 feedback signals from 12 macrocells as shown in Figure 27 and Figure 28. These local feedback signals can only feed into the macrocells of that quadrants and cannot be accessed by the other quadrants. The local bus has two kinds, one is from the Global Macrocell I/O Architecture output, and the other is from the Local Macrocell feedback. The local bus from the Local Macrocell feedback has two sources, one is from the I/O Architecture output and the other is from the I/O pin. Which one of the two sources to use is decided by the *feedback select switch* as shown in Figure 27. This switch is configured by the JEDEC file data.

The global bus can be divided into two classes, one is for the Global Macrocell feedbacks and the other is for global dedicated input pins. The Global Macrocell feedbacks come from each quadrant's 4 Global Macrocells as shown in Figure 26. Since there are four quadrants, the total global feedbacks are 16. The global bus can be accessed by any macrocells within the EP1800.

The EP1800 bus signal assignments are done at two entity levels. The global bus signals are assigned at the EP1800 entity level as shown in the EP1800 architecture box of Appendix A. The local bus signals are assigned at the Quadrant entity level as shown in the Quadrant architecture body of Appendix A. The global bus and local bus are concatenated together at the Quadrant entity level as follows

```
inputs <= local_bus & global_bus;
```

where the symbol "&" means concatenation. Here, this statement will connect the second array "global_bus" right after the first array, and form a new array "inputs". This new array will be fed to the macrocells.

G. UP-DOWN COUNTER

An application simulation of the EP1800 model was done in this study. The model was implemented as a 16 bits up/down counter. The counter was constructed by cascading two 8-bit counters together. The 8-bit counter used here was adopted from the ALTERA's TTL Macrofunction library [Ref. 8]. Figure 31 shows the 16-bit up-down counter block diagram and its corresponding function table.

The simulation was done by running a top entity discussed in Chapter II called the "TEST_BENCH". The code of the "TEST_BENCH" and the corresponding result are in Appendix B. Basically, the "TEST_BENCH" will call the EP1800 chip and provide the signals and the external JEDEC file name via the port map and the generic map to the EP1800. The JEDEC file can be created by any EPLD development tool as discussed in Chapter III.

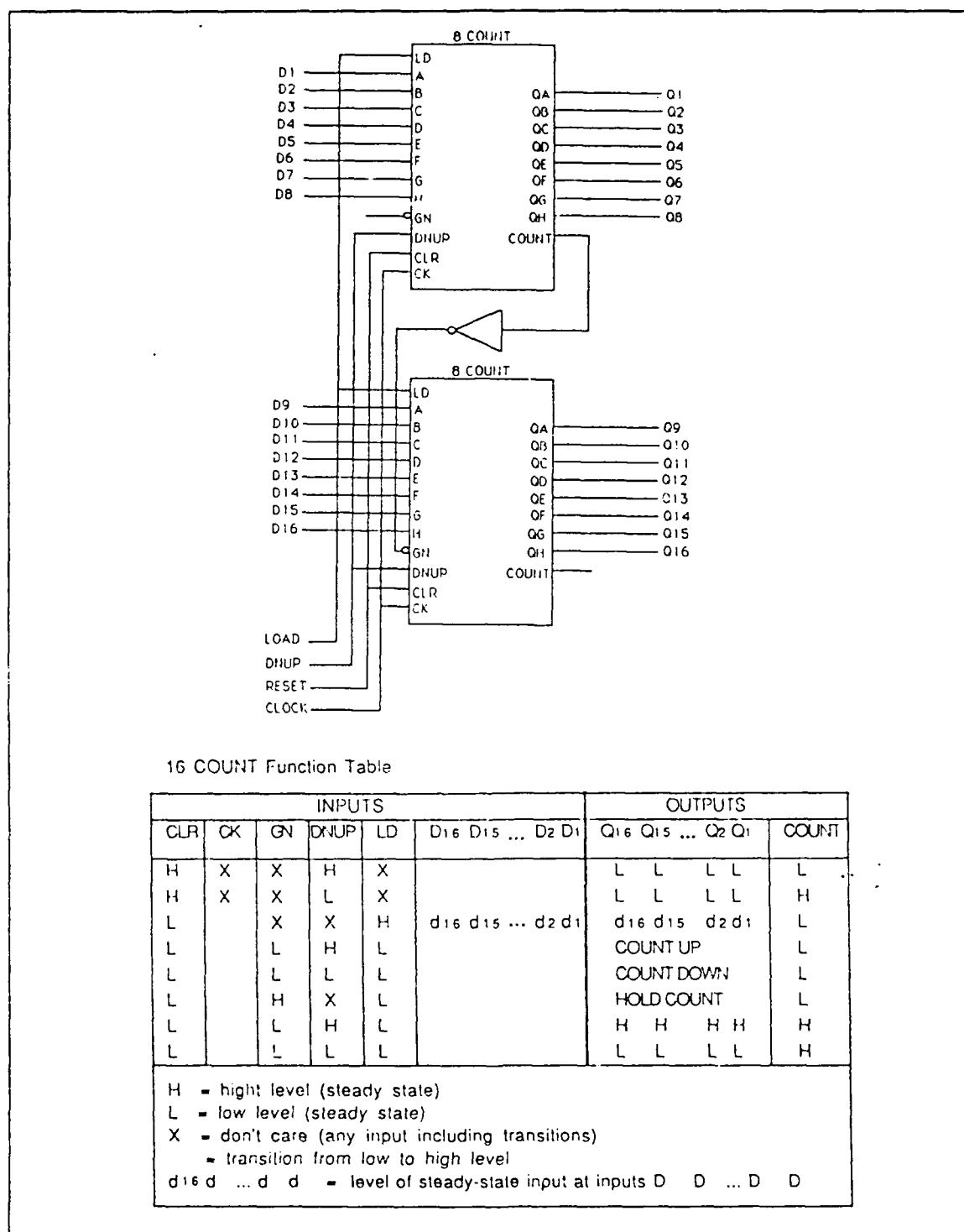


Figure 31. 16-bit up-down counter block diagram

V. CONCLUSIONS

A. GENERAL

In this thesis the general VHDL language features were introduced first. Then, the discussion of the EP310 and EP1800 structures leads to the modeling of these devices. Study revealed a number of problems in modeling these EPLDs. The solution of these problems were described and tested. The modeling technique used for EP310 and EP1800 can be equally applied to the modeling of other types of EPLD with similar structure.

It is very difficult to verify the model's accuracy. One way to verify the accuracy of the model is by running the manufacture's hardware test vectors for the corresponding hardware [Ref. 9]. Since the manufacture's test vector is not available, in this study only the correctness of the individual modules are tested. If all the modules can work accurately, the whole model can also work accurately.

The EP1800 and the EP1810 have a similar structure. The only difference between them is that the general macrocells in an EP1800 were replaced by the Enhanced Macrocells in the EP1810. The model of EP1800 with different timing parameters can also be used as the EP1810 model.

After using the VHDL to model the EPLD hardware, there are a few comments about the VHDL language:

1. The VHDL language is powerful. The VHDL has a very rich instruction set, and it also has the timing features in it. This makes the VHDL not only a very good hardware description language but also a very powerful simulation language.
2. The VHDL is very flexible. The language allows the users to declare their own object types and signal attributes, which makes the modeling easier.
3. The VHDL language can describe the hardware from three different views: behavioral, data-flow, and structural. The first two views lets the user model the hardware at an abstraction level. The hardware physical description is done at the structure level.

4. A hardware description can be detailed and accurate. The control statement and timing facility discussed in Chapter II allows the model to reflect not only the function of the hardware but also the timing characteristics.
5. The VHDL language statement is easy to comprehend. Since the VHDL is originally used as a description language, it allows meaningful variable and signal declarations. It also allows modules to be declared as functions. This make the program code more readable than most of the other languages.
6. The VHDL language learning curve is longer than those of the other languages. As was discussed before, the VHDL has a very rich instruction set. This sometimes is not a merit to the user, since rich instruction set means user need to take a longer time to learn more language features and rules.

B. PROGRAM SPEED

Like the other hardware simulation, the VHDL simulation takes a lot of time. In VHDL each simulation cycle is called a *delta cycle*. The simulation is event-driven, which means whenever any signal changes its value, the simulator will execute the corresponding statements once. If the model is too complicated or contains too many details which used up more internal signals, it will costs a lot of the CPU time. On the other hand, if the model is very abstract, i.e., modeling at the chip level or system level, then the simulation time can be reduced considerably.

The models built in this study are gate level models which contain a detailed circuit description. This means that the models built here need more execution time than the other approaches. Besides long simulation time, model analyzing, generating and building processes also takes longer time than anticipated.

The method of simulation developed in this work may not be the optimal one. One possible way to improve the simulation efficiency is to make the product-term algorithm more efficient. It is at the bottom of the hierarchy of the EPLD model and has the highest rate of execution.

In order to save programming time and reduce programming errors, the modeler should thoroughly decompose the target circuit and establish the interface between all modules via entity declaration before actually programming the architecture body.

C. RECOMMENDATIONS FOR FUTURE STUDY

The study of this thesis is concerated on the EPLD gate level modeling. But, the VHDL modeling technique and the modules built in this study can be applied to other logic system designs. The future study can be directed toward establishing a EPLD component library to support the increasing EPLD implementation in designs.

APPENDIX A. VHDL SOURCE CODE FOR EP310 AND EP1800

MODELS

A. VHDL SOURCE CODE FOR EPLD_PACKAGE

```
package EPROM_PACK is
  type Tri is ('U','0','1');
  type tri_vec is array (NATURAL range<>) of Tri;

  function resolver(signal inputs: tri_vec) return Tri;

  subtype tri_state is resolver Tri;
  type tri_vector is array (NATURAL range<>) of tri_state;

  function check_Pterm(P_string
string;x
tri_vector) return BIT;
  function bit_to_tri(inbit: bit) return Tri_state;
  function tri_to_bit(inbit: Tri_state) return bit;
  function trivec_to_bitvec(inbits: tri_vector) return bit_vector;
  function bitvec_to_trivec(inbits: bit_vector) return tri_vector;

end EPROM_PACK;

package body EPROM_PACK is

  function resolver(signal inputs: tri_vec) return tri is
    variable resolved_value: tri:= 'U';
    variable flag: integer:= 0;
  begin
    for i in inputs'range loop
      if inputs(i)/='U' then
        flag:= flag+1;
        resolved_value:= inputs(i);
      end if;
    end loop;
    assert flag <= 1
    report "iopin bus collision."
    severity FAILURE;
    return resolved_value;
  end resolver;

  function check_Pterm(P_string
string;x
tri_vector) return BIT is
    variable P_loc:natural:=0;
    variable ou:bit:='1';
    variable i:positive:=1;
```

```

begin
  while i <= P_string'length loop
    if(P_string(i)='0' and P_string(i+1)='0') then
      ou:='0';
      exit;
    end if;
    if(P_string(i)='1' and P_string(i+1)='1') then
      P_loc:=P_loc+1;
    end if;
    -- select the true input.
    if(P_string(i)='1' and P_string(i+1)='0') and
      (x((i+1)/2)='U' or x((i+1)/2)='0') then
      ou:='0';
      exit;
    end if;
    -- select the complement input.
    if(P_string(i)='0' and P_string(i+1)='1') and
      (x((i+1)/2)='U' or x((i+1)/2)='1') then
      ou:='0';
      exit;
    end if;
    i:=i+2;
  end loop;
  if P_loc = ((P_string'length)/2) then
    ou:='1';
  end if;
  return ou;
end check_Pterm;

```

```

function bit_to_tri(inbit: bit) return Tri_state is
begin
  if inbit='1' then
    return '1';
  else
    return '0';
  end if;
end bit_to_tri;

```

```

function tri_to_bit(inbit: Tri_state) return bit is
begin
  if inbit='1' then
    return '1';
  else
    return '0';
  end if;
end tri_to_bit;

```

```

function trivec_to_bitvec(inbits: tri_vector) return bit_vector is
  variable local: bit_vector(inbits'range);
begin
  for i in inbits'range loop
    if inbits(i)='1' then
      local(i):='1';
    end if;
  end loop;
  return local;
end trivec_to_bitvec;

```

```

        else
            local(i):='0';
        end if;
    end loop;
    return local;
end trivec_to_bitvec;

function bitvec_to_trivec(inbits: bit_vector) return tri_vector is
    variable local: tri_vector(inbits'range);
begin
    for i in inbits'range loop
        if inbits(i)='1' then
            local(i):='1';
        else
            local(i):='0';
        end if;
    end loop;
    return local;
end bitvec_to_trivec;

end EPROM_PACK;

```

B. VHDL SOURCE CODE FOR EP310 MODEL

```
package EP310_PACK is
    subtype IO_string is string(1 to 7);
    -- IO_string 1 to 7 are switches inside the io architecture;

    type IO_array_type is array(1 to 8) of IO_string;

    subtype P_string is string( 1 to 36);

    type P_array_type is array(NATURAL range<>) of P_string;

    type EP310_TYPE is record
        P_array: P_array_type(1 to 74);
        IO_array: IO_array_type;
    end record;

    -- the timing data are for EP310.
    constant tin: TIME:=10 ns;
    constant tio: TIME:=2 ns;
    constant tlad: TIME:=27 ns;
    constant tod: TIME:=12 ns;
    constant tzx: TIME:=0 ns; -- tzx(here) = tzx(table)-tod;
    constant txz: TIME:=0 ns; -- txz(here) = txz(table)-tod;
    constant tsu: TIME:=10 ns;
    constant th: TIME:=10 ns;
    constant tch: TIME:=16 ns;
    constant tics: TIME:=4 ns;
    constant tfd: TIME:=5 ns;
    constant tclr: TIME:=33 ns;

    function READ_JEDEC(F_name
        string) return EP310_TYPE;

end EP310_PACK;

use STD.TEXTIO.all;
package body EP310_PACK is
    function READ_JEDEC(F_name
        string) return EP310_TYPE is
        file F: text is in F_name;
        variable temp: line;
        variable temp_char: character;
        variable IO_temp: string(1 to 2730);
        variable EP310_MAP: EP310_TYPE;
        variable flag: boolean :=true;
        variable GOOD,L_flag: boolean:=false;
        variable j,k: integer:=1;
    begin
        -- cut out the unwanted portion.
        while flag loop
            readline(F,temp);
            read(temp,temp_char);
            if(temp_char='*') then
```

```

        L_flag:=true;
    end if;
    if(temp_char='L' and L_flag) then
        flag:=false;
    end if;
    assert not endfile(F)
        report "The input file is not correct";
    end loop;
-- extract the bit map information.
    while not endfile(F) loop
        readline(F,temp);
        j:=temp.all'length;
        IO_temp(k to k+j-1):=temp.all;
        k:= k+j;
    end loop;
    for i in EP310_MAP.p_array'range loop
        EP310_MAP.p_array(i):=IO_temp(1+36*(i-1) to i*36);
    end loop;
    for i in EP310_MAP.IO_array'range loop
        EP310_MAP.IO_array(i):=IO_temp(2665+7*(i-1) to 2664+7*i);
    end loop;
    return EP310_MAP;

    end READ_JEDEC;
end EP310_PACK;
*****
library EP310LIB,SHU;
use EP310LIB.EP310_PACK.all,SHU.EPROM_PACK.all;
entity EP310 is
    generic (JEDEC: in string);
    port (pin_1,pin_2,pin_3,pin_4,pin_5,pin_6,pin_7,pin_8,pin_9,pin_11
        : in tri_state; pin_12,pin_13,pin_14,pin_15,pin_16,pin_17,
        pin_18,pin_19: inout tri_state );
end EP310;

library EP310LIB,SHU;
use EP310LIB.EP310_pack.all,SHU.EPROM_PACK.all;
architecture STRUCTURAL of EP310 is

    component MACROCELL
        generic(macro_P_array: P_array_type(1 to 9));
        port(a
            tri_vector( 1 to 18);
            or_out,en: out bit);
    end component;

    component P_term
        generic( P: P_string);
        port(x: in tri_vector(1 to 18); p_out: out bit);
    end component;

    component IO_control
        generic(IO: Io_string);
        port(clk,or_in,preset,clear
bit;
            io_pin

```

```

tri_state;
    output,feedback: out tri_state);
end component;

signal macro_in : tri_vector(1 to 18);
signal oeloc,orloc:bit_vector(1 to 8);
signal local,output,feedback:tri_vector( 1 to 8);
signal clear,preset:bit;

constant BIT_MAP: EP310_TYPE := READ_JEDEC(JEDEC);

--configuration specifications.
for all:MACROCELL use entity EP310LIB.MACROCELL(behavioral);
for all:P_term use entity EP310LIB.P_term(behavioral);
for all:IO_CONTROL use entity EP310LIB.IO_CONTROL(behavioral_1);
begin
    macro_in(1)<= pin_1 after tin;
    macro_in(2)<= pin_11 after tin;
    macro_in(3)<= pin_2 after tin;
    macro_in(4)<= feedback(1);
    macro_in(5)<= pin_3 after tin;
    macro_in(6)<= feedback(2);
    macro_in(7)<= pin_4 after tin;
    macro_in(8)<= feedback(3);
    macro_in(9)<= pin_5 after tin;
    macro_in(10)<= feedback(4);
    macro_in(11)<= pin_6 after tin;
    macro_in(12)<= feedback(5);
    macro_in(13)<= pin_7 after tin;
    macro_in(14)<= feedback(6);
    macro_in(15)<= pin_8 after tin;
    macro_in(16)<= feedback(7);
    macro_in(17)<= pin_9 after tin;
    macro_in(18)<= feedback(8);

    -- generate eight macrocells.
M: for i in 1 to 8 generate
macro:    MACROCELL generic map(BIT_MAP.P_array(1+9*(i-1) to 9*i))
        port map(macro_in,orloc(i),oeloc(i));
    end generate;

    -- produce D_register preset input.
P:  P_term generic map(BIT_MAP.P_array(73))
    port map(macro_in,preset);

    -- produce D_register reset input.
R:  P_term generic map(BIT_MAP.P_array(74))
    port map(macro_in,clear);

    -- generate eight IO_control element.
CON: for i in 1 to 8 generate

CONTROL:  IO_control generic map(BIT_MAP.IO_array(i))
        port map(tri_to_bit(pin_1'delayed(tics+tin)),
                orloc(i),preset,clear,local(i),output(i),
                feedback(i));
    end generate;
end;

```

```

end generate;

local(1)<= output(1) after tio+tin when oeloc(1)='1' else
    pin_19 after tio+tin;

local(2)<= output(2) after tio+tin when oeloc(2)='1' else
    pin_18 after tio+tin;

local(3)<= output(3) after tio+tin when oeloc(3)='1' else
    pin_17 after tio+tin;

local(4)<= output(4) after tio+tin when oeloc(4)='1' else
    pin_16 after tio+tin;

local(5)<= output(5) after tio+tin when oeloc(5)='1' else
    pin_15 after tio+tin;

local(6)<= output(6) after tio+tin when oeloc(6)='1' else
    pin_14 after tio+tin;

local(7)<= output(7) after tio+tin when oeloc(7)='1' else
    pin_13 after tio+tin;

local(8)<= output(8) after tio+tin when oeloc(8)='1' else
    pin_12 after tio+tin;

pin_19<= output(1) after tod when oeloc(1)='1' else
    'U' after tod when oeloc(1)='0' else
    'U';

pin_18<= output(2) after tod when oeloc(2)='1' else
    'U' after tod when oeloc(2)='0' else
    'U';

pin_17<= output(3) after tod when oeloc(3)='1' else
    'U' after tod when oeloc(3)='0' else
    'U';

pin_16<= output(4) after tod when oeloc(4)='1' else
    'U' after tod when oeloc(4)='0' else
    'U';

pin_15<= output(5) after tod when oeloc(5)='1' else
    'U' after tod when oeloc(5)='0' else
    'U';

pin_14<= output(6) after tod when oeloc(4)='1' else
    'U' after tod when oeloc(6)='0' else
    'U';

pin_13<= output(7) after tod when oeloc(7)='1' else
    'U' after tod when oeloc(7)='0' else
    'U';

pin_12<= output(8) after tod when oeloc(8)='1' else
    'U' after tod when oeloc(8)='0' else
    'U';

```

```

        'U';

end STRUCTURAL;
*****
library EP310LIB,SHU;
use EP310LIB.EP310_pack.all,SHU.EPROM_PACK.all;
entity MACROCELL is
    generic (macro_P_array: P_array_type(1 to 9));
    port(a
        tri_vector( 1 to 18); or_out,en: out bit);
end MACROCELL;

library EP310LIB,SHU;
use EP310LIB.EP310_pack.all,SHU.EPROM_PACK.all;
architecture behavioral of MACROCELL is
    component P_term
        generic(P: P_string);
        port(x
            tri_vector(1 to 18); p_out:out bit);
        end component;
    signal loc: bit_vector(1 to 8);
    signal en_loc: bit;
    for all: P_term use entity EP310LIB.p_term(behavioral);
begin
    -- generate 8 P_terms.
P:    for i in 1 to 8 generate
        element:P_term generic map(macro_P_array(i))
            port map(a,loc(i));
    end generate;

OE:    P_term generic map(macro_P_array(9))
        port map(a,en_loc);
    or_out<= '1' when loc(1)='1' or loc(2)='1' or loc(3)='1' or
        loc(4)='1' or loc(5)='1' or loc(6)='1' or
        loc(7)='1' or loc(8)='1' else
        '0';
    en<= en_loc after tzx;
end behavioral;
*****
library EP310LIB,SHU;
use EP310LIB.EP310_pack.all,SHU.EPROM_PACK.all;
entity io_control is
    generic(IO: IO_string:="0000000");
    port(clk,or_in,preset,clear
        bit:='0';io_pin: in tri_state:='U';
        output,feedback:out tri_state);
end io_control;

library EP310LIB,SHU;
use EP310LIB.EP310_pack.all,SHU.EPROM_PACK.all;
architecture BEHAVIORAL_1 of io_control is
    component D_register
        generic(IO: IO_string:="0000000");
        port(d,ck,preset,clear: in bit:='0';q: inout bit);
    end component;

```

```

    signal Q_loc:BIT:='0';
    for all:d_register use entity EP310LIB.d_register(behavioral_2);
begin
    D1:D_register generic map(IO)
        port map (or_in,clk,preset,clear,Q_loc);

    process(io_pin,or_in,Q_loc)
    begin
-- OUTPUT SELECT.
        if IO(1)='0' then
            output<= bit_to_tri(not or_in) ;
        elsif IO(2)='0' then
            output<= bit_to_tri(or_in) ;
        elsif IO(3)='0' then
            output<= bit_to_tri(not Q_loc) ;
        elsif IO(4)='0' then
            output<= bit_to_tri(Q_loc) ;
        else
            output<= 'U';
        end if;

-- FEEDBACK SELECT.
        if IO(5)='0' then
            feedback<=bit_to_tri(or_in) after tfd;
        elsif IO(6)='0' then
            feedback<=bit_to_tri(Q_loc) after tfd;
        elsif IO(7)='0' then
            feedback<=io_pin;
        else
            feedback<= 'U';
        end if;
    end process ;
end BEHAVIORAL_1;
*****
library EP310LIB,SHU;
use EP310LIB.EP310_PACK.all,SHU.EPROM_PACK.all;
entity D_register is
    generic(IO: IO_string:="1111111");
    port(d,ck,preset,clear
        bit:='0'; q:out bit);
end D_register;

library EP310LIB,SHU;
use EP310LIB.EP310_PACK.all,SHU.EPROM_PACK.all;
architecture BEHAVIORAL_2 of D_register is
begin
    EDGE_TRIGGERED_D:
    block ((ck='1' and not ck'stable) or clear='1')
        signal s: bit;
    begin

-- check setup time of D_register.
        assert ck'stable or (ck='0') or d'stable(tsu) or (clear='1') or
            (IO(3)='1' and IO(4)='1' and IO(6)='1')
            -- not( not ck'stable and (ck='1') and not d'stable(tsu)
            -- and (clear='0')) and (IO(3)='0' or IO(4)='0' or IO(6)='0')

```

```

report "Setup Time Failure."
severity FAILURE;

-- check hold time of d_register.
assert ck'delayed(th)'stable or (ck'delayed(th)='0') or
      d'stable(th) or (clear='1') or
      (IO(3)='1' and IO(4)='1' and IO(6)='1')
  -- not (not ck'delayed(th)'stable and (ck'delayed(th)='0') and
  -- not d'stable(th) and (clear='0') and (IO(3)='0' or
  -- IO(4)='0' or IO(6)='0'))
report "Hold Time Failure."
severity FAILURE;

-- check setup time of D_register.(preset)
assert ck'stable or (ck='0') or preset'stable(tsu) or (clear='1') or
      (IO(3)='1' and IO(4)='1' and IO(6)='1')
  -- not (not ck'stable and (ck='1') and not preset'stable(tsu)
  -- and (clear='0') and (IO(3)='0' or IO(4)='0' or IO(6)='0'))
report "Setup Time Failure."
severity FAILURE;

-- check hold time of d_register.(preset)
assert ck'delayed(th)'stable or (ck'delayed(th)='0') or
      preset'stable(th) or (clear='1') or
      (IO(3)='1' and IO(4)='1' and IO(6)='1')
  -- not (not ck'delayed(th)'stable and (ck'delayed(th)='0') and
  -- not preset'stable(th) and (clear='0') and (IO(3)='0' or
  -- IO(4)='0' or IO(6)='0'))
report "Hold Time Failure."
severity FAILURE;

-- check minimum pulse width of d_register.
assert ck'stable or (ck='1') or ck'delayed'stable(tch) or (IO(3)='1'
      and IO(4)='1' and IO(6)='1')
  -- not (not ck'stable and ck='1' and ck'delayed'stable(tch) and
  -- (IO(3)='0' or IO(4)='0' or IO(6)='0'))
report "Minimum Pulse Width Failure."
severity FAILURE;

s<= guarded '1' when (preset='1' and clear='0') else
    d when (clear='0' and preset='0' and ck='1' and not ck'stable)
    else
    '0' after tclr when clear='1' else
    s;
q<= s;
end block EDGE_TRIGGERED_D;
end BEHAVIORAL_2;
*****
library EP310LIB,SHU;
use EP310LIB.EP310_pack.all,SHU.EPROM_PACK.all;
entity P_term is
  generic(P: P_string);
  port(x: in tri_vector(1 to 18); p_out: out bit);
end P_term;

```

```

library EP310LIB,SHU;
use EP310LIB.EP310_pack.all,SHU.EPROM_PACK.all;
architecture behavioral of P_term is
begin
    process(x)
        variable c:BIT;
    begin
        c:=check_Pterm(P,x);
        p_out<= c after tlad;
    end process;
end behavioral;

```

C. VHDL MODEL FOR EP1800.

```
library SHU;
use STD.TEXTIO.all, SHU.EPROM_PACK.all;
package EP1800_pack is

    subtype IO_string is string(1 to 5);
    -- IO_string 1 to 7 are switches inside the io architecture;

    type IO_array_type is array(NATURAL range<>) of IO_string;

    subtype P_string is string( 1 to 88);
    type P_array_type is array(NATURAL range <>) of P_string;
    subtype input_line is tri_vector(1 to 44);

    type EP1800_TYPE is record
        P_array: P_array_type(1 to 480);
        IO_array: IO_array_type(1 to 48);
    end record;

    -- all the time constants are for ep1800-2;

    constant tin:TIME:=10 ns;
    constant tio:TIME:=15 ns; -- tio(here):= tin(table)+tio(table);
    constant tlad:TIME:=40 ns;
    constant tlade:TIME:=35 ns;
    constant tod:TIME:=15 ns;
    constant tzx:TIME:=15 ns;
    constant txz:TIME:=15 ns;
    constant tsu:TIME:=12 ns;
    constant th:TIME:=30 ns;
    constant tch:TIME:=24 ns;
    constant tic:TIME:=40 ns;
    constant tice:TIME:=35 ns;
    constant tics:TIME:=4 ns;
    constant tfd:TIME:=10 ns;
    constant tclr:TIME:=40 ns;
    constant tclre:TIME:= 35 ns;
    -- here we demonstrate overload function.(which with same function name
    -- but differt input type).
    function REVERSE(A
        P_array_type(1 to 120)) return p_array_type;
    function REVERSE(A
        IO_array_type) return IO_array_type;
    function FIND(A
        IO_string; position: in natural)
        return character;
    function READ_JEDEC(F_name
        string) return EP1800_TYPE;

end EP1800_PACK;

package body EP1800_PACK is
    function REVERSE(A
```

```

P_array_type(1 to 120)) return P_array_type is
    variable rev_array: p_array_type(1 to 120);
begin
    for i in 1 to 12 loop
        rev_array(121-10*(i) to 120-10*(i-1))
            :=A(1+10*(i-1) to 10*(i));
    end loop;
    return rev_array;
end reverse;

function REVERSE(A
IO_array_type) return IO_array_type is
    variable rev_array: IO_array_type(1 to A'length);
    variable c: positive:=1;
begin
    for i in A'reverse_range loop
        rev_array(c):= A(i);
        c:= c+1;
    end loop;
    return rev_array;
end reverse;

function FIND(A
IO_string; position: in natural)
    return character is
begin
    return A(position);
end FIND;

function READ_JEDEC(F_name
string) return EP1800_TYPE is
    file F: text is in F_name;
    variable temp: line;
    variable temp_char: character;
    variable IO_temp: string(1 to 42500);
    variable EP1800_MAP: EP1800_TYPE;
    variable flag: boolean :=true;
    variable GOOD,L_flag: boolean:=false;
    variable j,k: integer:=1;
begin
    -- cut out the unwanted portion.
    while flag loop
        readline(F,temp);
        read(temp,temp_char);
        if(temp_char='*') then
            L_flag:=true;
        end if;
        if(temp_char='L' and L_flag) then
            flag:=false;
        end if;
        assert not endfile(F)
            report "The input file is not correct";
    end loop;
    -- extract the bit map information.
    while not endfile(F) loop
        readline(F,temp);

```

```

        j:=temp.all'length;
        IO_temp(k to k+j-1):=temp.all;
        k:= k+j;
    end loop;
    for i in EP1800_MAP.p_array'range loop
        EP1800_MAP.p_array(i):=IO_temp(1+88*(i-1) to i*88);
    end loop;
    for i in EP1800_MAP.io_array'range loop
        EP1800_MAP.IO_array(i):=IO_temp(42241+5*(i-1) to 42240+ i*5);
    end loop;
    return EP1800_MAP;

    end READ_JEDEC;
end EP1800_pack;
*****
library EP1800LIB, SHU;
use EP1800LIB.EP1800_PACK.all, SHU.EPROM_PACK.all;
entity EP1800 is
    generic ( JEDEC : in string);
    port (pin_14,pin_15,pin_16,pin_17,
          pin_19,pin_20,pin_21,pin_22,
          pin_48,pin_49,pin_50,pin_51,
          pin_53,pin_54,pin_55,pin_56
    tri_state:='U';
          pin_2,pin_3,pin_4,pin_5,pin_6,pin_7,
          pin_8,pin_9,pin_10,pin_11,pin_12,pin_13,
          pin_23,pin_24,pin_25,pin_26,pin_27,pin_28,
          pin_29,pin_30,pin_31,pin_32,pin_33,pin_34,
          pin_36,pin_37,pin_38,pin_39,pin_40,pin_41,
          pin_42,pin_43,pin_44,pin_45,pin_46,pin_47,
          pin_57,pin_58,pin_59,pin_60,pin_61,pin_62,
          pin_63,pin_64,pin_65,pin_66,pin_67,pin_68
          : inout tri_state:='U');
end EP1800;

library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
architecture STRUCTURAL of EP1800 is

    component QUADRANT
        generic(Q_P_array: P_array_type(1 to 120);
               Q_IO_array: IO_array_type(1 to 12));
        port(global_bus: in tri_vector(13 to 44);
              quad_clk: in bit;
              io_1,io_2,io_3,io_4,io_5,io_6,io_7,
              io_8: inout tri_state;
              quad: out tri_vector(1 to 4));
    end component;

    signal global_bus: tri_vector(13 to 44)
        :="UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU";
    signal quad_A,quad_B,quad_C,quad_D: tri_vector(1 to 4);
    constant BIT_MAP: EP1800_TYPE:= READ_JEDEC(JEDEC);

--configuration specification.

```

```

    for all: QUADRANT use entity EP1800LIB.QUADRANT(STRUCTURAL);
begin
-- Due to the proprietary reason, the global bus assignment statements
-- were removed.
    pin_10<= quad_A(1) after tod;-- (pin_10);
    pin_11<= quad_A(2) after tod;-- (pin_11);
    pin_12<= quad_A(3) after tod;-- (pin_12);
    pin_13<= quad_A(4) after tod;-- (pin_13);
    pin_26<= quad_B(1) after tod;-- (pin_26);
    pin_25<= quad_B(2) after tod;-- (pin_25);
    pin_24<= quad_B(3) after tod;-- (pin_24);
    pin_23<= quad_B(4) after tod;-- (pin_23);
    pin_44<= quad_C(1) after tod;-- (pin_44);
    pin_45<= quad_C(2) after tod;-- (pin_45);
    pin_46<= quad_C(3) after tod;-- (pin_46);
    pin_47<= quad_C(4) after tod;-- (pin_47);
    pin_60<= quad_D(1) after tod;-- (pin_60);
    pin_59<= quad_D(2) after tod;-- (pin_59);
    pin_58<= quad_D(3) after tod;-- (pin_58);
    pin_57<= quad_D(4) after tod;-- (pin_57);

QA:    QUADRANT
        generic map(BIT_MAP.P_array(1 to 120),
                     BIT_MAP.IO_array(1 to 12))
        port map(global_bus,tri_to_bit(pin_17),pin_2,pin_3,
                  pin_4,pin_5,pin_6,pin_7,pin_8,pin_9,
                  quad_A);

QB:    QUADRANT
        generic map(reverse(BIT_MAP.P_array(121 to 240)),
                     reverse(BIT_MAP.IO_array(13 to 24)))
        port map(global_bus,tri_to_bit(pin_19),pin_34,pin_33,
                  pin_32,pin_31,pin_30,pin_29,pin_28,pin_27,
                  quad_B);

QC:    QUADRANT
        generic map(BIT_MAP.P_array(241 to 360),
                     BIT_MAP.IO_array(25 to 36))
        port map(global_bus,tri_to_bit(pin_51),pin_36,pin_37,
                  pin_38,pin_39,pin_40,pin_41,pin_42,pin_43,
                  quad_C);

QD:    QUADRANT
        generic map(reverse(BIT_MAP.P_array(361 to 480)),
                     reverse(BIT_MAP.IO_array(37 to 48)))
        port map(global_bus,tri_to_bit(pin_53),pin_68,pin_67,
                  pin_66,pin_65,pin_64,pin_63,pin_62,pin_61,
                  quad_D);

end STRUCTURAL;
*****
library EP1800LIB, SHU;
use EP1800LIB.EP1800_PACK.all, SHU.EPROM_PACK.all;
entity QUADRANT is
    generic(Q_P_array: P_array_type(1 to 120);
            Q_IO_array: IO_array_type(1 to 12));

```

```

    port (global_bus: in tri_vector(13 to 44);
          quad_clk: in bit;
          io_1,io_2,io_3,io_4,io_5,io_6,io_7,
          io_8: inout tri_state;
          quad: out tri_vector(1 to 4));
end QUADRANT;

library EP1800LIB, SHU;
use EP1800LIB.EP1800_PACK.all, SHU.EPROM_PACK.all;
architecture STRUCTURAL of QUADRANT is

    component LOCAL_MACROCELL
        generic(P_array: P_array_type(1 to 10); IO: IO_string;
               t_lad: TIME:=tlad; t_clr: TIME:=tclr;
               t_ic: TIME:=tic);
        port(inputs
input_line; quad_clk: in bit;
          oe,output: out bit);
    end component;

    component GLOBAL_MACROCELL
        generic(P_array: P_array_type(1 to 10); IO: IO_string;
               t_lad: time:=tlade; t_clr: time:=tclr;
               t_ic: time:=tic);
        port(inputs
input_line; quad_clk: in bit;
          oe,output: out bit);
    end component;

    signal oe,M_output: bit_vector(1 to 12);
    signal local_bus: tri_vector( 1 to 12);
    signal local: tri_vector( 1 to 8);
    signal inputs: input_line;
    for all: LOCAL_MACROCELL
        use entity EP1800LIB.LOCAL_MACROCELL(structural);

    for all: GLOBAL_MACROCELL
        use entity EP1800LIB.GLOBAL_MACROCELL(structural`

begin

    inputs<= local_bus & global_bus;

M1_E:    LOCAL_MACROCELL generic map(Q_P_array(1 to 10),Q_IO_array(1),
                                   tlade,tclr,tice)
        port map (inputs,quad_clk,
                 oe(1),M_output(1));

M2_E:    LOCAL_MACROCELL generic map(Q_P_array(11 to 20),Q_IO_array(2),
                                   tlade,tclr,tice)
        port map (inputs,quad_clk,
                 oe(2),M_output(2));

M3_E:    LOCAL_MACROCELL generic map(Q_P_array(21 to 30),Q_IO_array(3),

```

```

                                tlade,tclre,tice)
                                port map (inputs,quad_clk,
                                              oe(3),M_output(3));

M4_E:    LOCAL_MACROCELL generic map(Q_P_array(31 to 40),Q_IO_array(4),
                                tlade,tclre,tice)
                                port map (inputs,quad_clk,
                                              oe(4),M_output(4));

M5_GE:    LOCAL_MACROCELL generic map(Q_P_array(41 to 50),Q_IO_array(5))
                                port map (inputs,quad_clk,
                                              oe(5),M_output(5));

M6_GE:    LOCAL_MACROCELL generic map(Q_P_array(51 to 60),Q_IO_array(6))
                                port map (inputs,quad_clk,
                                              oe(6),M_output(6));

M7_GE:    LOCAL_MACROCELL generic map(Q_P_array(61 to 70),Q_IO_array(7))
                                port map (inputs,quad_clk,
                                              oe(7),M_output(7));

M8_GE:    LOCAL_MACROCELL generic map(Q_P_array(71 to 80),Q_IO_array(8))
                                port map (inputs,quad_clk,
                                              oe(8),M_output(8));

M9_GL:    GLOBAL_MACROCELL generic map(Q_P_array(81 to 90),Q_IO_array(9))
                                port map (inputs,quad_clk,
                                              oe(9),M_output(9));

M10_GL:   GLOBAL_MACROCELL
                                generic map(Q_P_array(91 to 100),Q_IO_array(10))
                                port map (inputs,quad_clk,
                                              oe(10),M_output(10));

M11_GL:   GLOBAL_MACROCELL
                                generic map(Q_P_array(101 to 110),Q_IO_array(11))
                                port map (inputs,quad_clk,
                                              oe(11),M_output(11));

M12_GL:   GLOBAL_MACROCELL
                                generic map(Q_P_array(111 to 120),Q_IO_array(12))
                                port map (inputs,quad_clk,
                                              oe(12),M_output(12));

io_1<= bit_to_tri(M_output(1)) after tod when oe(1)='1' else
        'U' after tod when oe(1)='0' else
        'U';

io_2<= bit_to_tri(M_output(2)) after tod when oe(2)='1' else
        'U' after tod when oe(2)='0' else
        'U';

io_3<= bit_to_tri(M_output(3)) after tod when oe(3)='1' else
        'U' after tod when oe(3)='0' else
        'U';

```

```

io_4<= bit_to_tri(M_output(4)) after tod when oe(4)='1' else
      'U' after tod when oe(4)='0' else
      'U';

io_5<= bit_to_tri(M_output(5)) after tod when oe(5)='1' else
      'U' after tod when oe(5)='0' else
      'U';

io_6<= bit_to_tri(M_output(6)) after tod when oe(6)='1' else
      'U' after tod when oe(6)='0' else
      'U';

io_7<= bit_to_tri(M_output(7)) after tod when oe(7)='1' else
      'U' after tod when oe(7)='0' else
      'U';

io_8<= bit_to_tri(M_output(8)) after tod when oe(8)='1' else
      'U' after tod when oe(8)='0' else
      'U';

local(1)<= bit_to_tri(M_output(1)) after tio when oe(1)='1' else
          io_1 after tio;

local(2)<= bit_to_tri(M_output(2)) after tio when oe(2)='1' else
          io_2 after tio;

local(3)<= bit_to_tri(M_output(3)) after tio when oe(3)='1' else
          io_3 after tio;

local(4)<= bit_to_tri(M_output(4)) after tio when oe(4)='1' else
          io_4 after tio;

local(5)<= bit_to_tri(M_output(5)) after tio when oe(5)='1' else
          io_5 after tio;

local(6)<= bit_to_tri(M_output(6)) after tio when oe(6)='1' else
          io_6 after tio;

local(7)<= bit_to_tri(M_output(7)) after tio when oe(7)='1' else
          io_7 after tio;

local(8)<= bit_to_tri(M_output(8)) after tio when oe(8)='1' else
          io_8 after tio;

local_bus(1)<= local(1) when find(Q_IO_array(1),4)='0' else
              bit_to_tri(M_output(1)) after tfd;

local_bus(2)<= local(2) when find(Q_IO_array(2),4)='0' else
              bit_to_tri(M_output(2)) after tfd;

local_bus(3)<= local(3) when find(Q_IO_array(3),4)='0' else
              bit_to_tri(M_output(3)) after tfd;

local_bus(4)<= local(4) when find(Q_IO_array(4),4)='0' else
              bit_to_tri(M_output(4)) after tfd;

```

```

local_bus(5)<= local(5) when find(Q_IO_array(5),4)='0' else
    bit_to_tri(M_output(5)) after tfd;

local_bus(6)<= local(6) when find(Q_IO_array(6),4)='0' else
    bit_to_tri(M_output(6)) after tfd;

local_bus(7)<= local(7) when find(Q_IO_array(7),4)='0' else
    bit_to_tri(M_output(7)) after tfd;

local_bus(8)<= local(8) when find(Q_IO_array(8),4)='0' else
    bit_to_tri(M_output(8)) after tfd;

local_bus(9)<= bit_to_tri(M_output(9)) after tfd;
local_bus(10)<= bit_to_tri(M_output(10)) after tfd;
local_bus(11)<= bit_to_tri(M_output(11)) after tfd;
local_bus(12)<= bit_to_tri(M_output(12)) after tfd;

quad(1)<= bit_to_tri(M_output(9)) when oe(9)='1' else
    'U' when oe(9)='0' else
    'U';

quad(2)<= bit_to_tri(M_output(10)) when oe(10)='1' else
    'U' when oe(10)='0' else
    'U';

quad(3)<= bit_to_tri(M_output(11)) when oe(11)='1' else
    'U' when oe(11)='0' else
    'U';

quad(4)<= bit_to_tri(M_output(12)) when oe(12)='1' else
    'U' when oe(12)='0' else
    'U';

end STRUCTURAL;
*****
library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
entity LOCAL_MACROCELL is
    generic(P_array: P_array_type(1 to 10); IO: IO_string;
        t_lad:time:=tlad; t_clr:time:=tclr;
        t_ic:time:=tic);
    port(inputs
        input_line; quad_clk
        bit;
        oe,output:out bit);
end LOCAL_MACROCELL;

library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
architecture STRUCTURAL of LOCAL_MACROCELL is

    component P_term
        generic(P:P_string; t_lad:time:=tlad);
        port(x
            input_line; p_out:out bit);
    end component;

```

```

        component IO_CONTROL
            generic(I0: IO_string; t_clr: time:=tclr);
            port(or_in,clk,clear
bit;
            output: out bit);
        end component;

        signal local: bit_vector(1 to 10);
        signal or_out,clk: bit;

        for all: P_term use entity EP1800LIB.P_term(behavioral);
        for all: IO_CONTROL use entity EP1800LIB.IO_CONTROL(behavioral);
begin
P1:    P_term generic map(P_array(1),t_lad)
        port map(inputs,local(1));
P2:    P_term generic map(P_array(2),t_lad)
        port map(inputs,local(2));
P3:    P_term generic map(P_array(3),t_lad)
        port map(inputs,local(3));
P4:    P_term generic map(P_array(4),t_lad)
        port map(inputs,local(4));
P5:    P_term generic map(P_array(5),t_lad)
        port map(inputs,local(5));
P6:    P_term generic map(P_array(6),t_lad)
        port map(inputs,local(6));
P7:    P_term generic map(P_array(7),t_lad)
        port map(inputs,local(7));
P8:    P_term generic map(P_array(8),t_lad)
        port map(inputs,local(8));
CLR_P: P_term generic map(P_array(9),t_lad)
        port map(inputs,local(9));
OECK_P: P_term generic map(P_array(10),t_ic)
        port map(inputs,local(10));

        or_out<='1' when (local(1) or local(2) or local(3)
            or local(4) or local(5) or local(6)
            or local(7) or local(8))='1' else
            '0';

OECK_S: process(quad_clk,local(10))
begin
    if IO(2)='0' then
        clk<= quad_clk after tics;
        oe<= local(10) after tcz;
    else
        clk<= local(10);
        oe<='1' after tcz;
    end if;
end process OECK_S;

IO1:    IO_CONTROL generic map(I0,t_clr)
        port map(or_out,clk,local(9),output);

end STRUCTURAL;
*****

```

```

library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
entity GLOBAL_MACROCELL is
    generic(P_array:P_array_type(1 to 10); IO:IO_string;
           t_lad:time:=tlad; t_clr:time:=tclr;
           t_ic:time:=tic);
    port(inputs
         input_line; quad_clk
         bit;
           oe,output:out bit);
end GLOBAL_MACROCELL;

library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
architecture STRUCTURAL of GLOBAL_MACROCELL is

    component P_term
        generic(P:P_string; t_lad:time:=tlad);
        port(x: in input_line; p_out: out bit);
    end component;

    component IO_CONTROL
        generic(IO:IO_string; t_clr:time:=tclr);
        port(or_in,clk,clear: in bit;
             output: out bit);
    end component;

    signal local: bit_vector(1 to 10);
    signal or_out,clk,oeck: bit;

    for all: P_term use entity EP1800LIB.P_term(behavioral);
    for all: IO_CONTROL use entity EP1800LIB.IO_CONTROL(behavioral);
begin
P1:    P_term generic map(P_array(1))
        port map(inputs,local(1));
P2:    P_term generic map(P_array(2))
        port map(inputs,local(2));
P3:    P_term generic map(P_array(3))
        port map(inputs,local(3));
P4:    P_term generic map(P_array(4))
        port map(inputs,local(4));
P5:    P_term generic map(P_array(5))
        port map(inputs,local(5));
P6:    P_term generic map(P_array(6))
        port map(inputs,local(6));
P7:    P_term generic map(P_array(7))
        port map(inputs,local(7));
P8:    P_term generic map(P_array(8))
        port map(inputs,local(8));
CLR_P: P_term generic map(P_array(9))
        port map(inputs,local(9));
OECK_P:P_term generic map(P_array(10),tic)
        port map(inputs,local(10));

    or_out<='1' when (local(1) or local(2) or local(3)
                     or local(4) or local(5) or local(6)

```

```

        or local(7) or local(8)) ='1' else
        '0';

OECK_S: process(quad_clk,local(10))
begin
    if IO(2)='0' then
        clk<= quad_clk after tics;
        oeck<= local(10);
    else
        clk<= local(10);
        oeck<='1';
    end if;
end process OECK_S;

OE_S:   process(oeck)
begin
    if IO(4)='0' then
        oe<= oeck;
    else
        oe<= '0';
    end if;
end process OE_S;

IO_1:   IO_CONTROL generic map(IO)
        port map(or_out,clk,local(9),output);

end STRUCTURAL;
*****
library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
entity IO_CONTROL is
    generic(IO:string; t_clr:time:=tclr);
    port(or_in,clk,clear
        bit; output:out bit);
end IO_CONTROL;

-- For architecture body source code please contact Prof. CHIN-HWA LEE,
-- Naval Postgraduate School, Monterey, CA, 93943.

*****
library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
entity D_register is
    generic(IO:character='0'; t_clr:time:=tclr);
    port(d,clk,clear
        bit='0'; q:out bit);
end D_register;

library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
architecture BEHAVIORAL of D_register is
begin
    EDGE_TRIGGERED_D:
    block ((clk='1' and not clk'stable) or clear='1')
        signal s: bit;
    begin

```

```

-- check next setup time of D_register.
  assert clk'stable or (clk='0') or d'stable(tsu) or
    (clear='1') or IO='1'
    -- not( not clk'stable and (clk='1') and not d'stable(tsu)
    -- and (clear='0')) and IO='0' )
  report "Setup Time Fialure."
  severity FAILURE;

-- check hold time of d_register.
  assert clk'delayed(th)'stable or (clk'delayed(th)='0') or
    d'stable(th) or
    (clear='1') or IO='1'
    -- not (not clk'delayed(th)'stable and (clk'delared(th)='0') and
    -- not d'stable(th) and (clear='0') and IO='0' )
  report "Hold Time Failure."
  severity FAILURE;

-- check minimum pulse-width of d_register.
  assert clk'stable or (clk='1') or clk'delayed'stable(tch) or IO='1'
  -- not (clk'stable and (clk='1' and clk'delayed'stable(tch) and
  -- IO='0'))
  report "Minium pulse width failure"
  severity FAILURE;

  s<= guarded '0' after t_clr when clear='1' else
  d when (clear='0' and clk='1' and not clk'stable) else
    s;
  q<= s;

end block EDGE_TRIGGERED_D;
end BEHAVIORAL;
*****
library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
entity P_term is
  generic(P:P_string; t_lad:time:=tlad);
  port(x
    input_line; p_out:out bit);
end P_term;

library EP1800LIB, SHU;
use EP1800LIB.EP1800_pack.all, SHU.EPROM_PACK.all;
architecture behavioral of P_term is
begin
  process(x)
    variable c:BIT;
  begin
    c:=check_Pterm(P,x);
    p_out<= c after t_lad;
  end process;
end behavioral;
*****

```

APPENDIX B. VHDL CODE FOR TEST_BENCH

A. VHDL SOURCE CODE FOR TOP ENTITY DECLARATION.

```
-- NOTE: In the top-level design unit there can not have TIME generic
-- parameter, other wise there will have a error message in model
-- generate(MG) state and the MG process will stop without creating any
-- object file.
```

```
entity TEST_BENCH is
  generic (ck_rate: integer:= 20000000;
           term_sim: integer:= 10;
           delay: integer:=1000); -- delay unit ns.
end TEST_BENCH;
```

B. TEST_BENCH ARCHITECTURE BODY FOR EP310

```
library EP310LIB,SHU;
use EP310LIB.ep310_pack.all,SHU.EPROM_PACK.all;
architecture ep310 of test_bench is
    component ep310 generic(JEDEC: in string);
        port(pin_1,pin_2,pin_3,pin_4,pin_5,pin_6,pin_7,
            pin_8,pin_9,pin_11: in tri_state;
            pin_12,pin_13,pin_14,pin_15,pin_16,pin_17,
            pin_18,pin_19: inout tri_state );
    end component;

    signal pin_1,pin_2,pin_3,pin_4,pin_5,pin_6,
        pin_7,pin_8,pin_9,pin_11: tri_state:= 'U';
    signal pin_12,pin_13,pin_14,pin_15,pin_16,
        pin_17,pin_18,pin_19: resolver tri_state:= 'U';
    signal input: tri_vector( 1 to 10);
    signal io: tri_vector( 1 to 8);
    signal count: integer :=0;
    signal clock: bit:='0';

    for all : ep310 use entity EP310LIB.ep310(structural);
begin

    EP1:EP310 generic map("cntr7.jed")
        port map(pin_1,pin_2,pin_3,pin_4,pin_5,pin_6,pin_7,
            pin_8,pin_9,pin_11,
            pin_12,pin_13,pin_14,pin_15,pin_16,pin_17,
            pin_18,pin_19);

    CLOCK_GENERATOR:process(clock)
    begin
        clock<= not clock after 1 sec / ck_rate;
        pin_1<= bit_to_tri(clock);
    end process CLOCK_GENERATOR;

    pin_3<='1';          --after 2 sec /ck_rate; ENABLE.
    pin_2<='0'; --after 1 sec/ck_rate; RESET.

    TERMINATE:process(count)
    begin
        assert (count /= term_sim)
            report "simulation is done.";
    end process TERMINATE;

    IO_INPUT:block(clock = '1')
    begin
        pin_19<= guarded '1';
    end block IO_INPUT;

    count<= count+1 after delay*ns;

    input<= pin_1 & pin_2 & pin_3 & pin_3 & pin_5 & pin_6 & pin_7
```

```
        & pin_8 & pin_9 & pin_11;  
io <= pin_12 & pin_13 & pin_14 & pin_15 & pin_16 & pin_17 & pin_18  
    & pin_19;  
end ep310;
```

Vhdl Simulation Report

Kernel Library Name: <<SHU>>TEST_EP3

Kernel Creation Date: DEC-05-1988

Kernel Creation Time: 17:11:16

Run Identifier: 1

Run Date: DEC-05-1988

Run Time: 17:11:16

Report Control Language File: TEST_EP310.RCL

Report Output File : TEST_EP3.RPT

Max Time: 9223372036854775807

Max Delta: 2147483646

Report Control Language :

```
simulation_report s is
begin
    page_width is 72;
    select_signal : "c"=>pin_19;
    select_signal : "clk"=>pin_1;
    select_signal : pin_2,pin_3;
    select_signal : "pin_12 - pin_18"=> io(1 to 7);
    sample_signals by_event in ns;
end;
```

Report Format Information :

Time is in NS relative to the start of simulation
Time period for report is from 0 NS to End of Simulation
Signal values are reported by event (' ' indicates no event)

TIME	-----SIGNAL NAMES-----				
(NS)	c	clk	PIN_2	PIN_3	pin_12 - pin_18(1 TO 7)
0	'U'	'U'	'U'	'U'	"UUUUUUU"
+1		'0'	'0'	'1'	
39					
+1					"0000000"
50					
+1	'1'	'1'			
76					
+1					"0000001"
100					
+1		'0'			
150					
+1		'1'			
176					
+1					"0000010"
200					
+1		'0'			
250					
+1		'1'			
276					
+1					"0000011"
300					
+1		'0'			
350					
+1		'1'			
376					
+1					"0000100"
400					
+1		'0'			
450					
+1		'1'			
476					
+1					"0000101"
500					
+1		'0'			
550					
+1		'1'			
576					
+1					"0000110"
600					
+1		'0'			
650					
+1		'1'			
676					
+1					"0000111"
700					
+1		'0'			
750					
+1		'1'			
776					

+1 |
800 |
+1 |

'0'

"0001000"

C. TEST_BENCH ARCHITECTURE BODY FOR EP1800

```
library EP1800LIB, SHU;
use EP1800LIB.EP1800_PACK.all, SHU.EPROM_PACK.all;
architecture ep1800 of test_bench is
    component EP1800
        generic (JEDEC : in string);
        port (pin_14,pin_15,pin_16,pin_17,
            pin_19,pin_20,pin_21,pin_22,
            pin_48,pin_49,pin_50,pin_51,
            pin_53,pin_54,pin_55,pin_56
tri_state='0';
            pin_2,pin_3,pin_4,pin_5,pin_6,pin_7,
            pin_8,pin_9,pin_10,pin_11,pin_12,pin_13,
            pin_23,pin_24,pin_25,pin_26,pin_27,pin_28,
            pin_29,pin_30,pin_31,pin_32,pin_33,pin_34,
            pin_36,pin_37,pin_38,pin_39,pin_40,pin_41,
            pin_42,pin_43,pin_44,pin_45,pin_46,pin_47,
            pin_57,pin_58,pin_59,pin_60,pin_61,pin_62,
            pin_63,pin_64,pin_65,pin_66,pin_67,pin_68
            : inout tri_state='0');
    end component;

    signal DNUP,CLOCK,LOAD,RESET
        : tri_state='0';
    signal Q: tri_vector(1 to 16):="0000000000000000";
    signal D: tri_vector( 1 to 16):="0000000000000000";
    signal ck: bit :='0';
    signal c: positive:=1;
-- configuration specification
    for all : ep1800 use entity EP1800LIB.ep1800(structural);
begin

-- use named association interface list. associated lists are according to
-- chip map from ALTERA Design Processor Utilization Report.

    EP1:EP1800 generic map("count.jed")
        port map(pin_2=>Q(9),pin_3=>Q(10),pin_4=>Q(11),
            pin_5=>Q(12),pin_6=>Q(13),pin_7=>Q(14),
            pin_8=>Q(15),pin_9=>Q(16),pin_10=>Q(1),
            pin_11=>Q(2),pin_12=>Q(3),pin_13=>Q(7),
            pin_16=>DNUP,pin_17=>CLOCK,pin_19=>D(8),
            pin_20=>D(9),pin_21=>D(10),pin_22=>D(11),
            pin_23=>D(1),pin_24=>D(2),pin_25=>D(3),
            pin_26=>D(7),pin_48=>D(12),pin_49=>D(13),
            pin_50=>D(14),pin_51=>D(15),pin_53=>CLOCK,
            pin_54=>D(16),pin_55=>LOAD,pin_56=>RESET,
            pin_57=>Q(4),pin_58=>Q(5),pin_59=>Q(6),
            pin_60=>Q(8),pin_66=>D(4),pin_67=>D(5),
            pin_68=>D(6));

    CLOCK_GENERATOR: process(ck)
    begin
        ck<= not ck after delay*ns;
```

```

        CLOCK<= bit_to_tri(ck);
end process CLOCK_GENERATOR;

DNUP <= '1';
RESET <= '1' when c = 1 else
    '0';
LOAD <= '1' when c = 2 else
    '0';
c<= c+1 after delay*ns;

TERMINATE:process(ck)
begin
    assert (NOW /= term_sim*delay*ns)
        report "simulation is done.";
end process TERMINATE;

```

Vhdl Simulation Report

Kernel Library Name: <<SHU>>TEST_COUNT

Kernel Creation Date: NOV-24-1988

Kernel Creation Time: 14:37:26

Run Identifier: 1

Run Date: NOV-24-1988

Run Time: 14:37:26

Report Control Language File: TEST_EP1800.RCL

Report Output File : TEST_COUNT.RPT

Max Time: 9223372036854775807

Max Delta: 2147483646

Report Control Language :

```
simulation_report COUNT_UP is
begin
    page_width is 72;
    select_signal : "RE"=>RESET;
    select_signal : "clk"=>CLOCK;
    select_signal : "LO"=>LOAD;
    select_signal : "DNUP"=> DNUP;
    select_signal : Q;
    sample_signals by_event in ns;
end;
```

Report Format Information :

Time is in NS relative to the start of simulation

Time period for report is from 0 NS to End of Simulation

Signal values are reported by event (' ' indicates no event)

TIME	-----SIGNAL NAMES-----				
(NS)	RE	clk	LO	DNUP	Q(1 TO 16)
0	'0'	'0'	'0'	'0'	"UUUUUUUUUUUUUUUU"
+1	'1'			'1'	
55					"00000000UUUUUUUU"
70					
+1					"000000000000UUUU"
75					"0000000000000000"
1000					
+1	'0'	'1'	'1'		
2000					
+1		'0'	'0'		
3000					
+1		'1'			
3019					"1000000000000000"
4000					
+1		'0'			
5000					
+1		'1'			
5019					"0100000000000000"
6000					
+1		'0'			
7000					
+1		'1'			
7019					"1100000000000000"
8000					
+1		'0'			
9000					
+1		'1'			
9019					"0010000000000000"
10000					
+1		'0'			
11000					
+1		'1'			
11019					"1010000000000000"
12000					
+1		'0'			
13000					
+1		'1'			
13019					"0110000000000000"
14000					
+1		'0'			
15000					
+1		'1'			
15019					"1110000000000000"
16000					
+1		'0'			
17000					
+1		'1'			
17019					"0001000000000000"
18000					

+1 |
19000 |
+1 |
19019 |

'0'

'1'

"1001000000000000"

APPENDIX C. EXAMPLES OF SIGNAL SELECT FILE AND SIGNAL MAP FILE

A. SIGNAL SELECT FILE

```
-- selected trace signals for test_ep1800 are
: RESET;
: CLOCK;
: LOAD;
: DNUP;
: Q;
```

B. SIGNAL MAP OF THE TEST_EP3 MODEL

DEC-13-1988 14:26:12

VHDL Simulator
SIGNAL NAME MAP
KERNEL = <<SHU>>TEST_EP3

PAGE 1

```
: CLOCK ;
: COUNT ;
: INPUT(1 TO 10) ;
: IO(1 TO 8) ;
: PIN_1 ;
: PIN_11 ;
: PIN_12 ;
: PIN_13 ;
: PIN_14 ;
: PIN_15 ;
: PIN_16 ;
: PIN_17 ;
: PIN_18 ;
: PIN_19 ;
: PIN_2 ;
: PIN_3 ;
: PIN_4 ;
: PIN_5 ;
: PIN_6 ;
: PIN_7 ;
: PIN_8 ;
: PIN_9 ;
IO_INPUT : GUARD ;
/EP1 : CLEAR ;
/EP1 : FEEDBACK(1 TO 8) ;
/EP1 : LOCAL(1 TO 8) ;
/EP1 : MACRO_IN(1 TO 18) ;
/EP1 : OELOC(1 TO 8) ;
/EP1 : ORLOC(1 TO 8) ;
/EP1 : OUTPUT(1 TO 8) ;
/EP1 : PIN_1 ;
/EP1 : PIN_11 ;
/EP1 : PIN_12 ;
/EP1 : PIN_13 ;
/EP1 : PIN_14 ;
/EP1 : PIN_15 ;
/EP1 : PIN_16 ;
/EP1 : PIN_17 ;
/EP1 : PIN_18 ;
/EP1 : PIN_19 ;
/EP1 : PIN_2 ;
/EP1 : PIN_3 ;
/EP1 : PIN_4 ;
/EP1 : PIN_5 ;
/EP1 : PIN_6 ;
/EP1 : PIN_7 ;
/EP1 : PIN_8 ;
/EP1 : PIN_9 ;
/EP1 : PRESET ;
/EP1.CON(1) : CLK'TYPE_CONV_79_C ;
```

```

/EP1.CON(1) : PIN_1'DELAYED_79_A ;
/EP1.CON(1)/CONTROL : CLEAR ;
/EP1.CON(1)/CONTROL : CLK ;
/EP1.CON(1)/CONTROL : FEEDBACK ;
/EP1.CON(1)/CONTROL : IO_PIN ;
/EP1.CON(1)/CONTROL : OR_IN ;
DEC-13-1988 14:26:12

```

VHDL Simulator
SIGNAL NAME MAP
KERNEL = <<SHU>>TEST_EP3

PAGE 2

```

/EP1.CON(1)/CONTROL : OUTPUT ;
/EP1.CON(1)/CONTROL : PRESET ;
/EP1.CON(1)/CONTROL : Q_LOC ;
/EP1.CON(1)/CONTROL/D1 : CK ;
/EP1.CON(1)/CONTROL/D1 : CLEAR ;
/EP1.CON(1)/CONTROL/D1 : D ;
/EP1.CON(1)/CONTROL/D1 : PRESET ;
/EP1.CON(1)/CONTROL/D1 : Q ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_5 ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7 ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7'STABLE_26_9 ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_D ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F'STABLE_43_H ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I'STABLE_52_K ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_13_1 ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_18_3 ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_35_B ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_52_L ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_59_M ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_18_2 ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_26_4 ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : GUARD ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_35_A ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_43_C ;
/EP1.CON(1)/CONTROL/D1.EDGE_TRIGGERED_D : S ;
/EP1.CON(2) : CLK'TYPE_CONV_79_C ;
/EP1.CON(2) : PIN_1'DELAYED_79_A ;
/EP1.CON(2)/CONTROL : CLEAR ;
/EP1.CON(2)/CONTROL : CLK ;
/EP1.CON(2)/CONTROL : FEEDBACK ;
/EP1.CON(2)/CONTROL : IO_PIN ;
/EP1.CON(2)/CONTROL : OR_IN ;
/EP1.CON(2)/CONTROL : OUTPUT ;
/EP1.CON(2)/CONTROL : PRESET ;
/EP1.CON(2)/CONTROL : Q_LOC ;
/EP1.CON(2)/CONTROL/D1 : CK ;
/EP1.CON(2)/CONTROL/D1 : CLEAR ;
/EP1.CON(2)/CONTROL/D1 : D ;
/EP1.CON(2)/CONTROL/D1 : PRESET ;
/EP1.CON(2)/CONTROL/D1 : Q ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_5 ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7 ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7'STABLE_26_9 ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_D ;

```

```

/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F'STABLE_43_H ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I'STABLE_52_K ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_13_1 ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_18_3 ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_35_B ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_52_L ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_59_M ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_18_2 ;

```

DEC-13-1988 14:26:12

VHDL Simulator

PAGE 3

SIGNAL NAME MAP

KERNEL = <<SHU>>TEST_EP3

```

/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_26_4 ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : GUARD ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_35_A ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_43_C ;
/EP1.CON(2)/CONTROL/D1.EDGE_TRIGGERED_D : S ;
/EP1.CON(3) : CLK'TYPE_CONV_79_C ;
/EP1.CON(3) : PIN_1'DELAYED_79_A ;
/EP1.CON(3)/CONTROL : CLEAR ;
/EP1.CON(3)/CONTROL : CLK ;
/EP1.CON(3)/CONTROL : FEEDBACK ;
/EP1.CON(3)/CONTROL : IO_PIN ;
/EP1.CON(3)/CONTROL : OR_IN ;
/EP1.CON(3)/CONTROL : OUTPUT ;
/EP1.CON(3)/CONTROL : PRESET ;
/EP1.CON(3)/CONTROL : Q_LOC ;
/EP1.CON(3)/CONTROL/D1 : CK ;
/EP1.CON(3)/CONTROL/D1 : CLEAR ;
/EP1.CON(3)/CONTROL/D1 : D ;
/EP1.CON(3)/CONTROL/D1 : PRESET ;
/EP1.CON(3)/CONTROL/D1 : Q ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_5 ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7 ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7'STABLE_26_9 ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_D ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F'STABLE_43_H ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I'STABLE_52_K ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_13_1 ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_18_3 ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_35_B ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_52_L ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_59_M ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_18_2 ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_26_4 ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : GUARD ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_35_A ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_43_C ;
/EP1.CON(3)/CONTROL/D1.EDGE_TRIGGERED_D : S ;
/EP1.CON(4) : CLK'TYPE_CONV_79_C ;
/EP1.CON(4) : PIN_1'DELAYED_79_A ;
/EP1.CON(4)/CONTROL : CLEAR ;

```

```

/EP1.CON(4)/CONTROL : CLK ;
/EP1.CON(4)/CONTROL : FEEDBACK ;
/EP1.CON(4)/CONTROL : IO_PIN ;
/EP1.CON(4)/CONTROL : OR_IN ;
/EP1.CON(4)/CONTROL : OUTPUT ;
/EP1.CON(4)/CONTROL : PRESET ;
/EP1.CON(4)/CONTROL : Q_LOC ;
/EP1.CON(4)/CONTROL/D1 : CK ;
/EP1.CON(4)/CONTROL/D1 : CLEAR ;
/EP1.CON(4)/CONTROL/D1 : D ;
/EP1.CON(4)/CONTROL/D1 : PRESET ;
/EP1.CON(4)/CONTROL/D1 : Q ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_5 ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7 ;
DEC-13-1988 14:26:12

```

VHDL Simulator
SIGNAL NAME MAP
KERNEL = <<SHU>>TEST_EP3

PAGE 4

```

/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7'STABLE_26_9 ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_D ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F'STABLE_43_H ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I'STABLE_52_K ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_13_1 ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_18_3 ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_35_B ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_52_L ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_59_M ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_18_2 ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_26_4 ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : GUARD ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_35_A ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_43_C ;
/EP1.CON(4)/CONTROL/D1.EDGE_TRIGGERED_D : S ;
/EP1.CON(5) : CLK'TYPE_CONV_79_C ;
/EP1.CON(5) : PIN_1'DELAYED_79_A ;
/EP1.CON(5)/CONTROL : CLEAR ;
/EP1.CON(5)/CONTROL : CLK ;
/EP1.CON(5)/CONTROL : FEEDBACK ;
/EP1.CON(5)/CONTROL : IO_PIN ;
/EP1.CON(5)/CONTROL : OR_IN ;
/EP1.CON(5)/CONTROL : OUTPUT ;
/EP1.CON(5)/CONTROL : PRESET ;
/EP1.CON(5)/CONTROL : Q_LOC ;
/EP1.CON(5)/CONTROL/D1 : CK ;
/EP1.CON(5)/CONTROL/D1 : CLEAR ;
/EP1.CON(5)/CONTROL/D1 : D ;
/EP1.CON(5)/CONTROL/D1 : PRESET ;
/EP1.CON(5)/CONTROL/D1 : Q ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_5 ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7 ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7'STABLE_26_9 ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_D ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F'STABLE_43_H ;

```

```

/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I'STABLE_52_K ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_13_1 ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_18_3 ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_35_B ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_52_L ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_59_M ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_18_2 ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_26_4 ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : GUARD ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_35_A ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_43_C ;
/EP1.CON(5)/CONTROL/D1.EDGE_TRIGGERED_D : S ;
/EP1.CON(6) : CLK'TYPE_CONV_79_C ;
/EP1.CON(6) : PIN_1'DELAYED_79_A ;
/EP1.CON(6)/CONTROL : CLEAR ;
/EP1.CON(6)/CONTROL : CLK ;
/EP1.CON(6)/CONTROL : FEEDBACK ;
DEC-13-1988 14:26:12

```

VHDL Simulator
SIGNAL NAME MAP
KERNEL = <<SHU>>TEST_EP3

PAGE 5

```

/EP1.CON(6)/CONTROL : IO_PIN ;
/EP1.CON(6)/CONTROL : OR_IN ;
/EP1.CON(6)/CONTROL : OUTPUT ;
/EP1.CON(6)/CONTROL : PRESET ;
/EP1.CON(6)/CONTROL : Q_LOC ;
/EP1.CON(6)/CONTROL/D1 : CK ;
/EP1.CON(6)/CONTROL/D1 : CLEAR ;
/EP1.CON(6)/CONTROL/D1 : D ;
/EP1.CON(6)/CONTROL/D1 : PRESET ;
/EP1.CON(6)/CONTROL/D1 : Q ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_5 ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7 ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7'STABLE_26_9 ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_D ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F'STABLE_43_H ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I'STABLE_52_K ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_13_1 ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_18_3 ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_35_B ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_52_L ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_59_M ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_18_2 ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_26_4 ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : GUARD ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_35_A ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_43_C ;
/EP1.CON(6)/CONTROL/D1.EDGE_TRIGGERED_D : S ;
/EP1.CON(7) : CLK'TYPE_CONV_79_C ;
/EP1.CON(7) : PIN_1'DELAYED_79_A ;
/EP1.CON(7)/CONTROL : CLEAR ;
/EP1.CON(7)/CONTROL : CLK ;
/EP1.CON(7)/CONTROL : FEEDBACK ;

```

```

/EP1.CON(7)/CONTROL : IO_PIN ;
/EP1.CON(7)/CONTROL : OR_IN ;
/EP1.CON(7)/CONTROL : OUTPUT ;
/EP1.CON(7)/CONTROL : PRESET ;
/EP1.CON(7)/CONTROL : Q_LOC ;
/EP1.CON(7)/CONTROL/D1 : CK ;
/EP1.CON(7)/CONTROL/D1 : CLEAR ;
/EP1.CON(7)/CONTROL/D1 : D ;
/EP1.CON(7)/CONTROL/D1 : PRESET ;
/EP1.CON(7)/CONTROL/D1 : Q ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_5 ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7 ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7'STABLE_26_9 ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_D ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F'STABLE_43_H ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I'STABLE_52_K ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_13_1 ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_18_3 ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_35_B ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_52_L ;

```

DEC-13-1988 14:26:12

VHDL Simulator

PAGE 6

SIGNAL NAME MAP

KERNEL = <<SHU>>TEST_EP3

```

/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_59_M ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_18_2 ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_26_4 ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : GUARD ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_35_A ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_43_C ;
/EP1.CON(7)/CONTROL/D1.EDGE_TRIGGERED_D : S ;
/EP1.CON(8) : CLK'TYPE_CONV_79_C ;
/EP1.CON(8) : PIN_1'DELAYED_79_A ;
/EP1.CON(8)/CONTROL : CLEAR ;
/EP1.CON(8)/CONTROL : CLK ;
/EP1.CON(8)/CONTROL : FEEDBACK ;
/EP1.CON(8)/CONTROL : IO_PIN ;
/EP1.CON(8)/CONTROL : OR_IN ;
/EP1.CON(8)/CONTROL : OUTPUT ;
/EP1.CON(8)/CONTROL : PRESET ;
/EP1.CON(8)/CONTROL : Q_LOC ;
/EP1.CON(8)/CONTROL/D1 : CK ;
/EP1.CON(8)/CONTROL/D1 : CLEAR ;
/EP1.CON(8)/CONTROL/D1 : D ;
/EP1.CON(8)/CONTROL/D1 : PRESET ;
/EP1.CON(8)/CONTROL/D1 : Q ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_5 ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7 ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_26_7'STABLE_26_9 ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_D ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_43_F'STABLE_43_H ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'DELAYED_52_I'STABLE_52_K ;

```

```

/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_13_1 ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_18_3 ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_35_B ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_52_L ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : CK'STABLE_59_M ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_18_2 ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : D'STABLE_26_4 ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : GUARD ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_35_A ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : PRESET'STABLE_43_C ;
/EP1.CON(8)/CONTROL/D1.EDGE_TRIGGERED_D : S ;
/EP1.M(1)/MACRO : A(1 TO 18) ;
/EP1.M(1)/MACRO : EN ;
/EP1.M(1)/MACRO : EN_LOC ;
/EP1.M(1)/MACRO : LOC(1 TO 8) ;
/EP1.M(1)/MACRO : OR_OUT ;
/EP1.M(1)/MACRO.P(1)/ELEMENT : P_OUT ;
/EP1.M(1)/MACRO.P(1)/ELEMENT : X(1 TO 18) ;
/EP1.M(1)/MACRO.P(2)/ELEMENT : P_OUT ;
/EP1.M(1)/MACRO.P(2)/ELEMENT : X(1 TO 18) ;
/EP1.M(1)/MACRO.P(3)/ELEMENT : P_OUT ;
/EP1.M(1)/MACRO.P(3)/ELEMENT : X(1 TO 18) ;
/EP1.M(1)/MACRO.P(4)/ELEMENT : P_OUT ;
/EP1.M(1)/MACRO.P(4)/ELEMENT : X(1 TO 18) ;
/EP1.M(1)/MACRO.P(5)/ELEMENT : P_OUT ;
/EP1.M(1)/MACRO.P(5)/ELEMENT : X(1 TO 18) ;
DEC-13-1988 14:26:12      VHDL Simulator
                          SIGNAL NAME MAP

```

PAGE 7

KERNEL = <<SHU>>TEST_EP3

```

/EP1.M(1)/MACRO.P(6)/ELEMENT : P_OUT ;
/EP1.M(1)/MACRO.P(6)/ELEMENT : X(1 TO 18) ;
/EP1.M(1)/MACRO.P(7)/ELEMENT : P_OUT ;
/EP1.M(1)/MACRO.P(7)/ELEMENT : X(1 TO 18) ;
/EP1.M(1)/MACRO.P(8)/ELEMENT : P_OUT ;
/EP1.M(1)/MACRO.P(8)/ELEMENT : X(1 TO 18) ;
/EP1.M(1)/MACRO/OE : P_OUT ;
/EP1.M(1)/MACRO/OE : X(1 TO 18) ;
/EP1.M(2)/MACRO : A(1 TO 18) ;
/EP1.M(2)/MACRO : EN ;
/EP1.M(2)/MACRO : EN_LOC ;
/EP1.M(2)/MACRO : LOC(1 TO 8) ;
/EP1.M(2)/MACRO : OR_OUT ;
/EP1.M(2)/MACRO.P(1)/ELEMENT : P_OUT ;
/EP1.M(2)/MACRO.P(1)/ELEMENT : X(1 TO 18) ;
/EP1.M(2)/MACRO.P(2)/ELEMENT : P_OUT ;
/EP1.M(2)/MACRO.P(2)/ELEMENT : X(1 TO 18) ;
/EP1.M(2)/MACRO.P(3)/ELEMENT : P_OUT ;
/EP1.M(2)/MACRO.P(3)/ELEMENT : X(1 TO 18) ;
/EP1.M(2)/MACRO.P(4)/ELEMENT : P_OUT ;
/EP1.M(2)/MACRO.P(4)/ELEMENT : X(1 TO 18) ;
/EP1.M(2)/MACRO.P(5)/ELEMENT : P_OUT ;
/EP1.M(2)/MACRO.P(5)/ELEMENT : X(1 TO 18) ;
/EP1.M(2)/MACRO.P(6)/ELEMENT : P_OUT ;
/EP1.M(2)/MACRO.P(6)/ELEMENT : X(1 TO 18) ;
/EP1.M(2)/MACRO.P(7)/ELEMENT : P_OUT ;

```

```

/EP1.M(2)/MACRO.P(7)/ELEMENT : X(1 TO 18) ;
/EP1.M(2)/MACRO.P(8)/ELEMENT : P_OUT ;
/EP1.M(2)/MACRO.P(8)/ELEMENT : X(1 TO 18) ;
/EP1.M(2)/MACRO/OE : P_OUT ;
/EP1.M(2)/MACRO/OE : X(1 TO 18) ;
/EP1.M(3)/MACRO : A(1 TO 18) ;
/EP1.M(3)/MACRO : EN ;
/EP1.M(3)/MACRO : EN_LOC ;
/EP1.M(3)/MACRO : LOC(1 TO 8) ;
/EP1.M(3)/MACRO : OR_OUT ;
/EP1.M(3)/MACRO.P(1)/ELEMENT : P_OUT ;
/EP1.M(3)/MACRO.P(1)/ELEMENT : X(1 TO 18) ;
/EP1.M(3)/MACRO.P(2)/ELEMENT : P_OUT ;
/EP1.M(3)/MACRO.P(2)/ELEMENT : X(1 TO 18) ;
/EP1.M(3)/MACRO.P(3)/ELEMENT : P_OUT ;
/EP1.M(3)/MACRO.P(3)/ELEMENT : X(1 TO 18) ;
/EP1.M(3)/MACRO.P(4)/ELEMENT : P_OUT ;
/EP1.M(3)/MACRO.P(4)/ELEMENT : X(1 TO 18) ;
/EP1.M(3)/MACRO.P(5)/ELEMENT : P_OUT ;
/EP1.M(3)/MACRO.P(5)/ELEMENT : X(1 TO 18) ;
/EP1.M(3)/MACRO.P(6)/ELEMENT : P_OUT ;
/EP1.M(3)/MACRO.P(6)/ELEMENT : X(1 TO 18) ;
/EP1.M(3)/MACRO.P(7)/ELEMENT : P_OUT ;
/EP1.M(3)/MACRO.P(7)/ELEMENT : X(1 TO 18) ;
/EP1.M(3)/MACRO.P(8)/ELEMENT : P_OUT ;
/EP1.M(3)/MACRO.P(8)/ELEMENT : X(1 TO 18) ;
/EP1.M(3)/MACRO/OE : P_OUT ;
/EP1.M(3)/MACRO/OE : X(1 TO 18) ;
/EP1.M(4)/MACRO : A(1 TO 18) ;
/EP1.M(4)/MACRO : EN ;
DEC-13-1988 14:26:12

```

VHDL Simulator
SIGNAL NAME MAP
KERNEL = <<SHU>>TEST_EP3

PAGE 8

```

/EP1.M(4)/MACRO : EN_LOC ;
/EP1.M(4)/MACRO : LOC(1 TO 8) ;
/EP1.M(4)/MACRO : OR_OUT ;
/EP1.M(4)/MACRO.P(1)/ELEMENT : P_OUT ;
/EP1.M(4)/MACRO.P(1)/ELEMENT : X(1 TO 18) ;
/EP1.M(4)/MACRO.P(2)/ELEMENT : P_OUT ;
/EP1.M(4)/MACRO.P(2)/ELEMENT : X(1 TO 18) ;
/EP1.M(4)/MACRO.P(3)/ELEMENT : P_OUT ;
/EP1.M(4)/MACRO.P(3)/ELEMENT : X(1 TO 18) ;
/EP1.M(4)/MACRO.P(4)/ELEMENT : P_OUT ;
/EP1.M(4)/MACRO.P(4)/ELEMENT : X(1 TO 18) ;
/EP1.M(4)/MACRO.P(5)/ELEMENT : P_OUT ;
/EP1.M(4)/MACRO.P(5)/ELEMENT : X(1 TO 18) ;
/EP1.M(4)/MACRO.P(6)/ELEMENT : P_OUT ;
/EP1.M(4)/MACRO.P(6)/ELEMENT : X(1 TO 18) ;
/EP1.M(4)/MACRO.P(7)/ELEMENT : P_OUT ;
/EP1.M(4)/MACRO.P(7)/ELEMENT : X(1 TO 18) ;
/EP1.M(4)/MACRO.P(8)/ELEMENT : P_OUT ;
/EP1.M(4)/MACRO.P(8)/ELEMENT : X(1 TO 18) ;
/EP1.M(4)/MACRO/OE : P_OUT ;
/EP1.M(4)/MACRO/OE : X(1 TO 18) ;
/EP1.M(5)/MACRO : A(1 TO 18) ;

```

```

/EP1.M(5)/MACRO : EN ;
/EP1.M(5)/MACRO : EN_LOC ;
/EP1.M(5)/MACRO : LOC(1 TO 8) ;
/EP1.M(5)/MACRO : OR_OUT ;
/EP1.M(5)/MACRO.P(1)/ELEMENT : P_OUT ;
/EP1.M(5)/MACRO.P(1)/ELEMENT : X(1 TO 18) ;
/EP1.M(5)/MACRO.P(2)/ELEMENT : P_OUT ;
/EP1.M(5)/MACRO.P(2)/ELEMENT : X(1 TO 18) ;
/EP1.M(5)/MACRO.P(3)/ELEMENT : P_OUT ;
/EP1.M(5)/MACRO.P(3)/ELEMENT : X(1 TO 18) ;
/EP1.M(5)/MACRO.P(4)/ELEMENT : P_OUT ;
/EP1.M(5)/MACRO.P(4)/ELEMENT : X(1 TO 18) ;
/EP1.M(5)/MACRO.P(5)/ELEMENT : P_OUT ;
/EP1.M(5)/MACRO.P(5)/ELEMENT : X(1 TO 18) ;
/EP1.M(5)/MACRO.P(6)/ELEMENT : P_OUT ;
/EP1.M(5)/MACRO.P(6)/ELEMENT : X(1 TO 18) ;
/EP1.M(5)/MACRO.P(7)/ELEMENT : P_OUT ;
/EP1.M(5)/MACRO.P(7)/ELEMENT : X(1 TO 18) ;
/EP1.M(5)/MACRO.P(8)/ELEMENT : P_OUT ;
/EP1.M(5)/MACRO.P(8)/ELEMENT : X(1 TO 18) ;
/EP1.M(5)/MACRO/OE : P_OUT ;
/EP1.M(5)/MACRO/OE : X(1 TO 18) ;
/EP1.M(6)/MACRO : A(1 TO 18) ;
/EP1.M(6)/MACRO : EN ;
/EP1.M(6)/MACRO : EN_LOC ;
/EP1.M(6)/MACRO : LOC(1 TO 8) ;
/EP1.M(6)/MACRO : OR_OUT ;
/EP1.M(6)/MACRO.P(1)/ELEMENT : P_OUT ;
/EP1.M(6)/MACRO.P(1)/ELEMENT : X(1 TO 18) ;
/EP1.M(6)/MACRO.P(2)/ELEMENT : P_OUT ;
/EP1.M(6)/MACRO.P(2)/ELEMENT : X(1 TO 18) ;
/EP1.M(6)/MACRO.P(3)/ELEMENT : P_OUT ;
/EP1.M(6)/MACRO.P(3)/ELEMENT : X(1 TO 18) ;
/EP1.M(6)/MACRO.P(4)/ELEMENT : P_OUT ;
DEC-13-1988 14:26:12      VHDL Simulator
                          SIGNAL NAME MAP
                          KERNEL = <<SHU>>TEST_EP3

```

PAGE 9

```

/EP1.M(6)/MACRO.P(4)/ELEMENT : X(1 TO 18) ;
/EP1.M(6)/MACRO.P(5)/ELEMENT : P_OUT ;
/EP1.M(6)/MACRO.P(5)/ELEMENT : X(1 TO 18) ;
/EP1.M(6)/MACRO.P(6)/ELEMENT : P_OUT ;
/EP1.M(6)/MACRO.P(6)/ELEMENT : X(1 TO 18) ;
/EP1.M(6)/MACRO.P(7)/ELEMENT : P_OUT ;
/EP1.M(6)/MACRO.P(7)/ELEMENT : X(1 TO 18) ;
/EP1.M(6)/MACRO.P(8)/ELEMENT : P_OUT ;
/EP1.M(6)/MACRO.P(8)/ELEMENT : X(1 TO 18) ;
/EP1.M(6)/MACRO/OE : P_OUT ;
/EP1.M(6)/MACRO/OE : X(1 TO 18) ;
/EP1.M(7)/MACRO : A(1 TO 18) ;
/EP1.M(7)/MACRO : EN ;
/EP1.M(7)/MACRO : EN_LOC ;
/EP1.M(7)/MACRO : LOC(1 TO 8) ;
/EP1.M(7)/MACRO : OR_OUT ;
/EP1.M(7)/MACRO.P(1)/ELEMENT : P_OUT ;
/EP1.M(7)/MACRO.P(1)/ELEMENT : X(1 TO 18) ;

```

```

/EP1.M(7)/MACRO.P(2)/ELEMENT : P_OUT ;
/EP1.M(7)/MACRO.P(2)/ELEMENT : X(1 TO 18) ;
/EP1.M(7)/MACRO.P(3)/ELEMENT : P_OUT ;
/EP1.M(7)/MACRO.P(3)/ELEMENT : X(1 TO 18) ;
/EP1.M(7)/MACRO.P(4)/ELEMENT : P_OUT ;
/EP1.M(7)/MACRO.P(4)/ELEMENT : X(1 TO 18) ;
/EP1.M(7)/MACRO.P(5)/ELEMENT : P_OUT ;
/EP1.M(7)/MACRO.P(5)/ELEMENT : X(1 TO 18) ;
/EP1.M(7)/MACRO.P(6)/ELEMENT : P_OUT ;
/EP1.M(7)/MACRO.P(6)/ELEMENT : X(1 TO 18) ;
/EP1.M(7)/MACRO.P(7)/ELEMENT : P_OUT ;
/EP1.M(7)/MACRO.P(7)/ELEMENT : X(1 TO 18) ;
/EP1.M(7)/MACRO.P(8)/ELEMENT : P_OUT ;
/EP1.M(7)/MACRO.P(8)/ELEMENT : X(1 TO 18) ;
/EP1.M(7)/MACRO/OE : P_OUT ;
/EP1.M(7)/MACRO/OE : X(1 TO 18) ;
/EP1.M(8)/MACRO : A(1 TO 18) ;
/EP1.M(8)/MACRO : EN ;
/EP1.M(8)/MACRO : EN_LOC ;
/EP1.M(8)/MACRO : LOC(1 TO 8) ;
/EP1.M(8)/MACRO : OR_OUT ;
/EP1.M(8)/MACRO.P(1)/ELEMENT : P_OUT ;
/EP1.M(8)/MACRO.P(1)/ELEMENT : X(1 TO 18) ;
/EP1.M(8)/MACRO.P(2)/ELEMENT : P_OUT ;
/EP1.M(8)/MACRO.P(2)/ELEMENT : X(1 TO 18) ;
/EP1.M(8)/MACRO.P(3)/ELEMENT : P_OUT ;
/EP1.M(8)/MACRO.P(3)/ELEMENT : X(1 TO 18) ;
/EP1.M(8)/MACRO.P(4)/ELEMENT : P_OUT ;
/EP1.M(8)/MACRO.P(4)/ELEMENT : X(1 TO 18) ;
/EP1.M(8)/MACRO.P(5)/ELEMENT : P_OUT ;
/EP1.M(8)/MACRO.P(5)/ELEMENT : X(1 TO 18) ;
/EP1.M(8)/MACRO.P(6)/ELEMENT : P_OUT ;
/EP1.M(8)/MACRO.P(6)/ELEMENT : X(1 TO 18) ;
/EP1.M(8)/MACRO.P(7)/ELEMENT : P_OUT ;
/EP1.M(8)/MACRO.P(7)/ELEMENT : X(1 TO 18) ;
/EP1.M(8)/MACRO.P(8)/ELEMENT : P_OUT ;
/EP1.M(8)/MACRO.P(8)/ELEMENT : X(1 TO 18) ;
/EP1.M(8)/MACRO/OE : P_OUT ;
DEC-13-1988 14:26:12

```

VHDL Simulator
SIGNAL NAME MAP
KERNEL = <<SHU>>TEST_EP3

PAGE 10

```

/EP1.M(8)/MACRO/OE : X(1 TO 18) ;
/EP1/P : P_OUT ;
/EP1/P : X(1 TO 18) ;
/EP1/R : P_OUT ;
/EP1/R : X(1 TO 18) ;

```

APPENDIX D. MACRO VAX/VMS SYSTEM COMMAND

A. MACRO VAX/VMS SYSTEM COMMAND FOR EP310 MODEL

```
!FILE NAME: Batch (for EP310 model)
$set def [shu.vhdl.altera]
$set verify
$vlb setlib shu
$vhdl eprom_pack
$mg eprom_pack
$mg/body eprom_pack
$set def [shu.vhdl.altera.ep310]
$set verify
$vlb setlib ep310
$vhdl ep310_pack
$mg ep310_pack
$mg/body ep310_pack
$vhdl d_reg
$mg d_register(behavioral_2)
$vhdl p_term
$mg p_term(behavioral)
$vhdl macrocell
$mg macrocell(behavioral)
$vhdl io_control
$mg io_control(behavioral_1)
$vhdl ep310
$mg ep310(structural)
$set def [shu.vhdl.altera.ep310.test]
$set verify
$vlb setlib shu
$vhdl test_bench
$vhdl test_ep310
$mg/top test_bench(ep310)
$build/replace/ker=test_ep3 test_bench(ep310)
$sim test_ep3/param=20000000,4
$rg test_ep3 test_ep310.rcl
$exit
```

B. MACRO VAX/VMS SYSTEM COMMAND FOR EP1800 MODEL

```
!FILE NAME: Batch (for EP1800 model)
$set def [shu.vhdl.altera]
$set verify
$vls setlib shu
$vhdl eprom_pack
$mg eprom_pack
$mg/body eprom_pack
$set def [shu.vhdl.altera.ep1800]
$set verify
$vls setlib ep1800
$vhdl ep1800_pack
$mg ep1800_pack
$mg/body ep1800_pack
$vhdl d_reg
$mg d_register(behavioral)
$vhdl p_term
$mg p_term(behavioral)
$vhdl io_control
$mg io_control(behavioral)
$vhdl local_m
$mg local_macrocell(structural)
$vhdl global_m
$mg global_macrocell(structural)
$vhdl quadrant
$mg quadrant(structural)
$vhdl ep1800
$mg ep1800(structural)
$set def [shu.vhdl.altera.ep1800.test]
$set verify
$vls setlib shu
$vhdl test_bench
$vhdl test_ep1800
$mg/top test_bench(ep1800)
$build/replace/ker=test_count test_bench(ep1800)
$sim test_count/para=0,20/trace=select.signal
$rg test_count test_ep1800.rcl
$exit
```

LIST OF REFERENCES

1. "VHDL and System Design," in *Defense Science & Electronics*, Vol. 5, No. 10, pp. 49-52, October 1986.
2. J.R. Armstrong, "Chip-level modeling with HDLS," in *IEEE Design & Test of Computers*, pp. 8-18, February 1988.
3. James R. Armstrong, *Chip-level modeling with VHDL*, pp. 90-93, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
4. *IEEE Standard VHDL Language Reference Manual*, IEEE Inc., New York, 1987.
5. *User's Manual for the Standard VHDL 1076 Support Environment(draft)*, Intermetrics Inc., Bethesda, Maryland, 5 August 1988.
6. *Altera Databook second printing*, Altera Corporation, Santa Clara, California, January 1988.
7. "EPLD timing simulation," in *Altera User-configurable Logic Applications Handbook*, pp. 83-95, Altera Corporation, Santa Clara, California, July 1988.
8. "Counter Design," in *Altera User-configurable Logic Applications Handbook*, pp. 55-59, Altera Corporation, Santa Clara, California, January 1988.

9. Richard Goering, "Modeling strategies simplify board-level simulation," in *Computer Design*, pp. 29-33, March 1988.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor Chin-Hwa Lee, Code 62Le Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	20
5.	Professor Jon T. Butler, Code 62Bu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Jim Armstrong Virginia Tech, E.E. Dept Blacksburg, VA 24061	1
7.	Luis Concha Electronics Technology Laboratory WRDC/ELED WPAFB, OH 45433	1
8.	John W. Hines USAF WRDC/ELED WPAFB, OH 45433-6543	1
9.	Paul Hunter NRL Code 5305 Washington, DC 20375-506	1
10.	Kim Kanzaki AFIT/ENG WPAAFB WPAFB, OH 45433	1

- | | | |
|-----|---|---|
| 11. | Steven Levitan
Univ. of Pittsburgh
Dept.Elec
348 Benedum Hall
Pittsburgh, PA 15261 | 1 |
| 12. | Carl Schaefer
Intermetrics, Inc.
4733 Bethesda Ave.
Bethesda, MD 20815 | 1 |
| 13. | Ronald Waxman
Univ. of Virginia
Dept. of EE
Thornton Hall
Charlottesville, VA 22903 | 1 |